VISIT…

Advanced .NET Remoting, Second Edition

INGO RAMMER AND MARIO SZPUSZTA

Apress®

Advanced .NET Remoting, Second Edition

Copyright © 2005 by Ingo Rammer and Mario Szpuszta

All rights reserved. No part of this work may be reproduced or transmitted in any form or by any means, electronic or mechanical, including photocopying, recording, or by any information storage or retrieval system, without the prior written permission of the copyright owner and the publisher.

ISBN (pbk): 1-59059-417-7

Printed and bound in the United States of America 9 8 7 6 5 4 3 2 1

Trademarked names may appear in this book. Rather than use a trademark symbol with every occurrence of a trademarked name, we use the names only in an editorial fashion and to the benefit of the trademark owner, with no intention of infringement of the trademark.

Lead Editor: Ewan Buckingham
Technical Reviewer: Kent Sharkey
Editorial Board: Steve Anglin, Dan Appleman, Ewan Buckingham, Gary Cornell, Tony Davis, Jason Gilmore, Jonathan Hassell, Chris Mills, Dominic Shakeshaft, Jim Sumser
Project Manager: Laura E. Brown
Copy Manager: Nicole LeClerc
Copy Editor: Ami Knox
Production Manager: Kari Brooks-Copony
Production Editor: Ellie Fountain
Compositor: Kinetic Publishing Services, LLC
Proofreader: Elizabeth Berry
Indexer: John Collin
Artist: Kinetic Publishing Services, LLC
Cover Designer: Kurt Krames
Manufacturing Manager: Tom Debolski

Distributed to the book trade in the United States by Springer-Verlag New York, Inc., 233 Spring Street, 6th Floor, New York, NY 10013, and outside the United States by Springer-Verlag GmbH & Co. KG, Tiergartenstr. 17, 69112 Heidelberg, Germany.

In the United States: phone 1-800-SPRINGER, fax 201-348-4505, e-mail `orders@springer-ny.com`, or visit `http://www.springer-ny.com`. Outside the United States: fax +49 6221 345229, e-mail `orders@springer.de`, or visit `http://www.springer.de`.

For information on translations, please contact Apress directly at 2560 Ninth Street, Suite 219, Berkeley, CA 94710. Phone 510-549-5930, fax 510-549-5939, e-mail `info@apress.com`, or visit `http://www.apress.com`.

The information in this book is distributed on an “as is” basis, without warranty. Although every precaution has been taken in the preparation of this work, neither the author(s) nor Apress shall have any liability to any person or entity with respect to any loss or damage caused or alleged to be caused directly or indirectly by the information contained in this work.

The source code for this book is available to readers at `http://www.apress.com` in the Downloads section.

To Katja,
Who was courageous enough to marry me
even though she knew I would write another book.
—Ingo

To my parents—I am so happy that I have you!
And to my best friends Dominik and Edi—I enjoy every single moment with you!
—Mario

Contents at a Glance

About the Authors xvii
About the Technical Reviewer xix
Acknowledgments xxi
Introduction xxiii

PART 1 ■■■ Understanding

CHAPTER 1 Introduction to Remoting 3
CHAPTER 2 .NET Remoting Basics 9
CHAPTER 3 .NET Remoting in Action 25
CHAPTER 4 Configuration and Deployment 75
CHAPTER 5 Securing .NET Remoting 123
CHAPTER 6 Creating Remoting Clients 161
CHAPTER 7 In-Depth .NET Remoting 185
CHAPTER 8 The Ins and Outs of Versioning 225
CHAPTER 9 .NET Remoting Tips and Best Practices 275
CHAPTER 10 Troubleshooting .NET Remoting 303

PART 2 ■■■ Extending

CHAPTER 11 Inside the Framework 321
CHAPTER 12 Creation of Sinks 349
CHAPTER 13 Extending .NET Remoting 359
CHAPTER 14 Developing a Transport Channel 421
CHAPTER 15 Context Matters 469

PART 3 Reference

APPENDIX A .NET Remoting Usage Reference 487
APPENDIX B .NET Remoting Extensibility Reference 525
APPENDIX C .NET Remoting Links 541

INDEX 549

Contents

About the Authors xvii
About the Technical Reviewer xix
Acknowledgments xxi
Introduction xxiii

PART 1 ■■■ Understanding

CHAPTER 1 Introduction to Remoting 3

What Is Remoting? 3
Scenarios for .NET Remoting 3
Centralized Business Logic 4
Physical Separation of Layers 4
Accessing Other Platforms 4
Evolution of Remoting 4
DCE/RPC 5
CORBA 5
DCOM 5
MTS/COM+ 6
Java RMI 6
Java EJB 6
Web Services/SOAP/XML-RPC 7
.NET Remoting 7
Summary 7

CHAPTER 2 .NET Remoting Basics 9

Advantages of .NET Remoting 9
Ease of Implementation 9
Extensible Architecture 10
Interface Definitions 11
Serialization of Data 12
Lifetime Management 12
Multiserver/Multiclient 13

Your First Remoting Application . . . 13
The First Sample . . . 14
Extending the Sample . . . 20
Summary . . . 23

CHAPTER 3 .NET Remoting in Action . . . 25

Types of Remoting . . . 25
ByValue Objects . . . 25
MarshalByRefObjects . . . 26
Types of Invocation . . . 46
Synchronous Calls . . . 47
Asynchronous Calls . . . 51
One-Way Calls . . . 55
Multiserver Configuration . . . 59
Examining a Sample Multiserver Application . . . 60
Sharing Assemblies . . . 67
Shared Implementation . . . 67
Shared Interfaces . . . 67
Shared Base Classes . . . 67
SoapSuds-Generated Metadata . . . 68
Summary . . . 74

CHAPTER 4 Configuration and Deployment . . . 75

Configuration Files . . . 76
Watch for the Metadata! . . . 77
The Problem with SoapSuds . . . 77
Porting the Sample to Use Configuration Files . . . 82
Standard Configuration Options . . . 85
What About Interfaces? . . . 100
Using the IPC Channel in .NET Remoting 2.0 . . . 102
Deployment . . . 108
Console Applications . . . 108
Windows Services . . . 108
Deployment Using IIS . . . 116
Summary . . . 121

CHAPTER 5 Securing .NET Remoting 123

Building Secure Systems 123
Authentication Protocols in Windows 124
NTLM Authentication 124
Kerberos: Very Fast Track 126
Security Package Negotiate 128
Security Support Provider Interface 128
Identities and Principals: A Short Overview 129
Securing with IIS 133
Authentication with IIS 133
Encryption and IIS 138
Security Outside of IIS 140
Using the MSDN Security Samples 140
Implementing Authorization in the Server 149
Security with Remoting in .NET 2.0 (Beta) 151
Summary 160

CHAPTER 6 Creating Remoting Clients 161

Creating a Server for Your Clients 161
Creating a Console Client 163
Creating Windows Forms Clients 167
Creating Back-End–Based Clients 169
ASP.NET-Based Clients 169
Remoting Components Hosted in IIS As Clients 172
Security Considerations 177
Summary 184

CHAPTER 7 In-Depth .NET Remoting 185

Managing an Object's Lifetime 185
Understanding Leases 186
Working with Sponsors 196
Using the CallContext 209
Best Practices 212
Security and the Call Context 213
Remoting Events 213
Events: First Take 214
Refactoring the Event Handling 217
Why [OneWay] Events Are a Bad Idea 222
Summary 224

CHAPTER 8 The Ins and Outs of Versioning 225

.NET Framework Versioning Basics 225
A Short Introduction to Strong Naming 225
Versioning in .NET Remoting—Fundamentals 233
Versioning of Server-Activated Objects 233
Versioning of Client-Activated Objects 240
Versioning of [Serializable] Objects 242
Advanced Versioning Concepts 246
Versioning with Interfaces 246
Versioning Concepts for Serialized Types 256
Summary 273

CHAPTER 9 .NET Remoting Tips and Best Practices 275

.NET Remoting Use Cases 275
Cross-AppDomain Remoting 276
Cross-Process on a Single Machine 276
Cross-Process on Multiple Machines in a LAN 276
Cross-Process via WAN/Internet 278
Nonusage Scenarios 279
The Nine Rules of Scalable Remoting 280
Using Events and Sponsors 281
How to Notify Nevertheless 282
Message Queuing to the Rescue 283
Other Approaches 286
SoapSuds vs. Interfaces in .NET Remoting 286
Custom Exceptions 288
Scaling Out Remoting Solutions 290
Load Balancing Basics 290
Taking Nodes Online/Offline 299
Designing Applications for Static Scalability 299
Summary 301

CHAPTER 10 Troubleshooting .NET Remoting 303

Debugging Hints 303
Manual Breakpoints 304
Configuration File Settings 305
Local or Remote? 307
Checking Types on Your Server 308

BinaryFormatter Version Incompatibility 309
Troubleshooting with a Custom Sink 310
Changing Security Restrictions with TypeFilterLevel 311
Using Custom Exceptions 313
Multihomed Machines and Firewalls 315
Client-Activated Objects Behind Firewalls 317
Summary 318

PART 2 Extending

CHAPTER 11 Inside the Framework 321

Looking at the Five Elements of Remoting 321
A Bit About Proxies 322
Understanding the Role of Messages 326
Examining Message Sinks 328
Serialization Through Formatters 329
Moving Messages Through Transport Channels 330
Client-Side Messaging 331
ClientContextTerminatorSink and Dynamic Sinks 332
SoapClientFormatterSink 333
HttpClientChannel 333
Server-Side Messaging 333
HttpServerChannel and HttpServerTransportSink 335
SDLChannelSink 335
SoapServerFormatterSink and BinaryServerFormatterSink 336
DispatchChannelSink 336
CrossContextChannel 336
ServerContextTerminatorSink 337
LeaseSink 337
ServerObjectTerminatorSink and StackbuilderSink 337
All About Asynchronous Messaging 338
Asynchronous IMessageSink Processing 338
Asynchronous IClientChannelSink Processing 340
Generating the Request 342
Handling the Response 345
Server-Side Asynchronous Processing 347
Summary 348

CHAPTER 12 Creation of Sinks 349

Understanding Sink Providers 349
Creating Client-Side Sinks 350
Creating Server-Side Sinks 354
Using Dynamic Sinks 356
Summary 357

CHAPTER 13 Extending .NET Remoting 359

Creating a Compression Sink 359
Implementing the Client-Side Sink 361
Implementing the Server-Side Sink 364
Creating the Sink Providers 367
Using the Sinks 369
Extending the Compression Sink 371
Encrypting the Transfer 375
Essential Symmetric Encryption 376
Creating the Sinks 380
Creating the Providers 386
Passing Runtime Information 390
Changing the Programming Model 402
Using This Sink 408
Avoiding the BinaryFormatter Version Mismatch 409
Using a Custom Proxy 413
Some Final Words of Caution 419
Summary 419

CHAPTER 14 Developing a Transport Channel 421

Protocol Considerations 421
The Shortcut Route to SMTP 422
. . . And Round-Trip to POP3 423
Character Encoding Essentials 424
Creating E-Mail Headers 425
Encapsulating the Protocols 426
Checking for New Mail 433
Registering a POP3 Server 435
Connecting to .NET Remoting 437
Implementing the Client Channel 445
Creating the Client's Sink and Provider 449

Implementing the Server Channel 453
Creating the Server's Sink 458
Wrapping the Channel 462
Using the SmtpChannel 465
Preparing Your Machine 467
Some Final Words of Caution. 468
Summary 468

CHAPTER 15 Context Matters 469

Working at the MetaData Level. 471
Creating a Context. 472
Checking Parameters in an IMessageSink 480
Summary 483
Conclusion 484

PART 3 Reference

APPENDIX A .NET Remoting Usage Reference 487

System Types 487
System.Activator Class 488
System.MarshalByRefObject Class. 488
System.SerializableAttribute Class 489
System.Delegate Class 490
System.IAsyncResult Interface 491
System.Runtime.Remoting 491
Basic Infrastructure Classes 491
Configuration Classes 493
Exception Classes 497
General Interfaces 498
System.Runtime.Remoting.Channels. 499
General Interfaces and Classes 499
System.Runtime.Remoting.Channels.Http 504
HttpChannel Class 504
HttpClientChannel Class 505
HttpServerChannel Class 506
System.Runtime.Remoting.Channels.Tcp 506
TcpChannel Class 506
TcpClientChannel Class 507
TcpServerChannel Class. 508

System.Runtime.Remoting.Lifetime 508
ILease Interface 508
ISponsor Interface 509
ClientSponsor Class 510
LifetimeServices Class 511
LeaseState Enumeration 511
System.Runtime.Remoting.Messaging 512
AsyncResult Class 512
CallContext Class 512
LogicalCallContext Class 514
OneWayAttribute Class 514
System.Runtime.Remoting.Metadata 514
SoapAttribute Class 515
SoapTypeAttribute Class 515
SoapFieldAttribute Class 515
SoapMethodAttribute Class 516
SoapParameterAttribute Class 516
SoapOption Enumeration 516
System.Runtime.Remoting.Services 516
EnterpriseServicesHelper Class 516
RemotingClientProxy Class 517
ITrackingHandler Interface 517
TrackingServices Class 517
System.Runtime.Serialization 518
ISerializable Interface 519
SerializationInfo Class 520
StreamingContext Structure 520
SerializationException Class 521
System.Runtime.Serialization.Formatters 521
SoapFault Class 521
SoapMessage Class 521
TypeFilterLevel Enumeration 521
Summary 523

APPENDIX B .NET Remoting Extensibility Reference 525

System.Runtime.Remoting.Messaging 525
IMessage Interface 525
IMessageSink Interface 526
IMethodMessage Interface 527
IMethodCallMessage Interface 528

IMethodReturnMessage Interface 528
MethodCall Class 529
MethodResponse Class 529
System.Runtime.Remoting.Activation 529
IConstructionCallMessage Interface 530
IConstructionReturnMessage Interface 530
System.Runtime.Remoting.Proxies 530
RealProxy Class 531
ProxyAttribute Class 531
System.Runtime.Remoting.Channels 531
IChannelSinkBase Interface 532
IClientChannelSink Interface 532
IClientChannelSinkProvider Interface 533
IClientFormatterSink Interface 534
IClientFormatterSinkProvider Interface 534
IServerChannelSink Interface 534
IServerChannelSinkProvider Interface 535
ITransportHeaders Interface 536
IChannel Interface 537
IChannelReceiver Interface 538
IChannelSender Interface 539
BaseChannelObjectWithProperties Class 539
BaseChannelWithProperties Class 540
BaseChannelSinkWithProperties Class 540
Summary 540

APPENDIX C .NET Remoting Links 541

Ingo's .NET Remoting FAQ Corner 541
MSDN and MSDN Magazine Articles 541
"Improving Remoting Performance" 541
".NET Remoting Security" 541
"Boundaries: Processes and Application Domains" 542
".NET Remoting Architectural Assessment" 542
".NET Remoting Overview" 542
"Building Secure ASP.NET Applications: Authentication, Authorization, and Secure Communication" 542
"NET Remoting Authentication and Authorization Sample" 542
"Managed Extensions for C++ and .NET Remoting Tutorial" 543
".NET Remoting Use-Cases and Best Practices" and "ASP.NET Web Services or .NET Remoting: How to Choose" 543

"Remoting Examples" . . . 543
"Secure Your .NET Remoting Traffic by Writing an Asymmetric Encryption Channel" . . . 543
"Create a Custom Marshaling Implementation Using .NET Remoting and COM Interop" . . . 543
.NET Remoting Interoperability . . . 544
.NET Remoting: CORBA Interoperability . . . 544
.NET Remoting: Java RMI Bridges . . . 544
XML-RPC with .NET Remoting . . . 544
Custom .NET Remoting Channels . . . 544
Named Pipes Channel for .NET Remoting . . . 545
TcpEx Channel for .NET Remoting . . . 545
Jabber Channel . . . 545
Remoting Channel Framework Extension . . . 545
"Using MSMQ for Custom Remoting Channel" . . . 545
"Using WSE-DIME for Remoting over Internet" . . . 546
Interesting Technical Articles . . . 546
C# Corner: Remoting Section . . . 546
"Share the ClipBoard Using .NET Remoting" . . . 546
"Chaining Channels in .NET Remoting" . . . 546
"Applying Observer Pattern in .NET Remoting" . . . 546
"Aspect-Oriented Programming Enables Better Code Encapsulation and Reuse" and ".NET Remoting Spied On" . . . 547
"Persistent Events in Stateless Remoting Server" . . . 547
"Intrinsyc's Ja.NET—Extending the Reach of .NET Remoting" . . . 547
"Implementing Object Pooling with .NET Remoting—Part I" . . . 547
".NET Remoting Versus Web Services" . . . 547
".NET Remoting Central" . . . 547
"Output Caching for .NET Remoting" . . . 548
"Abstract Client Formatter Sink" . . . 548
Remoting Tools . . . 548
Remoting Management Console . . . 548
Remoting Probe . . . 548

INDEX . . . 549

About the Authors

INGO RAMMER is cofounder of thinktecture, a company supporting software architects and developers with architecture and design of .NET and Web Services applications. He is a regular speaker about these topics at conferences around the world, author of numerous online and print articles, and winner of the *.NET Developer's Journal*'s Readers' Choice Award for Best .NET Book of 2003. You can reach him at `http://www.thinktecture.com/staff/ingo`.

MARIO SZPUSZTA is working in the Developer and Platform Group of Microsoft Austria. Before he started working for Microsoft, Mario was involved in several projects based on COM+ and DCOM with Visual Basic and Visual C++ as well as projects based on Java and J2SE. With Beta 2 of the .NET Framework, he started developing Web applications with ASP.NET. Right now, as developer evangelist for Microsoft Austria, he is doing workshops, trainings, and proof-of-concept projects together with independent software vendors in Austria based on .NET, Web Services, and Office 2003 technologies.

About the Technical Reviewer

KENT SHARKEY is currently the content strategist for ASP.NET and Visual Studio content for MSDN. When not answering e-mail, he occasionally writes, codes, and sleeps. He lives in green, green Redmond with his wife, and two furry "children," Squirrel and Cica.

Acknowledgments

First and foremost, I want to thank Mario for taking the challenge to write the second edition of this book together with me. Without him, this book could not exist.

I especially want to thank those people whom I've met in the previous years and whose insight constantly influenced the way I think about software. A big "thank you" therefore needs to go to Beat Schwegler, Clemens Vasters, Don Box, and Steve Swartz.

But the biggest "thank you" needs to go to the thousands of software developers and architects who contacted me by e-mail, who chatted with me at various conferences, and who wrote thought-provoking Weblog and newsgroup posts. You are the reason why I wrote this book.

Of course, writing a book would not be possible without the support of many people. Even though only Mario's and my name appear on the cover, this book would not have been possible without my fellows at thinktecture: Christian Weyer, Christian Nagel, and Ralf Westphal; my friend and the technical reviewer for this book, Kent Sharkey; and the fine editorial staff at Apress: Laura Brown, Ami Knox, and Ellie Fountain. Thank you for making this book a reality!

—Ingo Rammer

The first large project I started working on was one of the most interesting projects I have ever been part of. The enthusiasm and creativity of the two masterminds in this team, Harald Leitenmüller and Benedikt Redl, have inspired me. These two persons have shown me what software development in large projects really means, and they have shown me how interesting software architecture is! Without them, I would not have progressed even half as far as I have today. Therefore, my biggest thanks go to Harald and Benedikt.

Without Beat Schwegler, I would not have had the chance to get involved in writing this book. Thank you, Beat, for this great opportunity and much more for the things I have learned in the past two years from you. These two years have been a really great time!

Last but not least, I want to thank you, Ingo, for giving me the chance to write this book with you. It was really great. I learned many things, and right now I recognize that writing concepts and thoughts down is something that can be really funny and, even more so, interesting.

—Mario Szpuszta

Introduction

In the time since the first edition of this book has been published, quite a bit has changed in the world of software development on Microsoft's platforms. The .NET Framework has become a tried-and-true development platform, and service orientation gained a larger-than-expected momentum. The latter especially presents a very challenging task for the developer using .NET Remoting: the need to avoid possible incompatibilities with future paradigms. If service orientation will, in the next few years, gain the success it deserves, it might be important for your application to be developed in a way to easily adopt these new ideas.

In this book, I have therefore followed a slightly different approach from the one I did in the previous edition. While the first book focused only on covering all the features of the .NET Remoting framework, Mario and I tried to extend this second edition with the best practices for using this technology. While it still covers nearly each and every feature of the .NET Remoting framework, the largest part of the new chapters of this book—especially Chapters 5, 8, 9, and 10—deals with security, best practices, and the general avoidance of problems.

What Is Covered in This Book

This book covers the means of cross-process and cross-machine interaction of applications developed with the .NET Framework. It will provide you with an in-depth understanding of the remoting capabilities that are built into the .NET Framework.

.NET Remoting is different from most other means of remote object access because it can be as easy as writing COM components in Visual Basic 6, yet also gives you the option to extend remoting to include virtually any protocol on any transportation layer you will come across.

Part 1 of the book gives you a thorough introduction to .NET Remoting basics and how you can use .NET Remoting "out of the box." This gives you a whole range of possibilities, from fast binary transfer protocol to a cross-platform SOAP protocol, with the potential to switch between both without changing a single line in your code. At the end of this part, you will be able to design and develop remoteable components and know just what you have to do to achieve your goals. This part also deals with objects' lifetimes, security, versioning, marshalling, and deployment.

Part 2 covers the advanced features of .NET Remoting and its extensibility model. At the end of the second part, you will have an in-depth understanding of the inner workings of remoting and will know how to extend the framework to meet your requirements. You should not be afraid, especially as you go through the sample code in the second part of the book, to either hit F1 or to insert a breakpoint and examine the Locals window in your custom channel sink to see the exact contents of the objects that get passed as parameters to your methods.

What This Book Doesn't Cover

This book is in no way a rehash of the supplied documentation, but is meant to be used in conjunction with it. You will only find a small percentage of the information that is covered in the online documentation in this book and vice versa, so it is very important for you to use the .NET Framework SDK documentation as well.

I chose this approach to writing a book for one simple reason: I assume that, as an advanced developer, you don't have much time to waste going through a 1,000-page book of which 600 pages are a reproduction of the online documentation. Instead, you want to read the information that has not been covered before. If you think so as well, this book is right for you.

Who This Book Is For

This book is for the intermediate-to-advanced programmer who wants a hands-on guide to .NET Remoting. Although this book is not an introduction to .NET, the CLR, or any .NET language, you nevertheless will be able to use the knowledge and insight you'll get from this book with any of these programming languages. All the samples printed in this book are written in Visual Basic .NET, but you can download each and every sample in both C# and Visual Basic .NET.

If you are a "use-it" developer, Part 1 (Chapters 1 through 10) of this book will serve you well by providing a general introduction to the possibilities of remoting and giving you in-depth information on how to use the capabilities that come with .NET Remoting "out of the box." This part also includes guidance on security, best practices, and troubleshooting.

If you are more of an "understand-it-and-extend-it" developer, Part 2 of this book is for you. Chapters 11 through 15 were written for those who want to understand what's going on behind the scenes of .NET Remoting and how the framework can be customized using proxies, messages, channel sinks, and providers. It also demonstrates how a complete transport channel is implemented from scratch.

At the end of the book, you'll find a collection of appendixes that provide a reference of the namespaces, classes, and interfaces that comprise the .NET Remoting framework.

How This Book Is Structured

Advanced .NET Remoting is divided into two parts. Part 1 (Chapters 1 through 10) covers everything you need to know for developing distributed applications within the .NET Framework. Part 2 (Chapters 11 through 15) gives you a thorough technical insight that will allow you to really understand what's happening behind the scenes and how you can tap into customizing the framework to suit your exact needs. Following is a brief chapter-by-chapter summary of the topics covered in this book.

Chapter 1: Introduction to Remoting

This chapter gives you a short introduction to the world of distributed application development and the respective technologies. It presents some scenarios in which .NET Remoting can be employed and includes historical background on the progress and development of various remoting frameworks during the last ten years.

Chapter 2: .NET Remoting Basics

This chapter gets you started with your first remoting application. Before going directly into the code, I present the distinctions between .NET Remoting and other distributed application frameworks. I then introduce you to the basic types of remote objects, which are server-activated objects and client-activated objects, and show you how to pass data by value. I also give you some basic information about lifetime management issues and the generation of metadata, which is needed for the client to know about the interfaces of the server-side objects.

Chapter 3: .NET Remoting in Action

In this chapter, I demonstrate the key techniques you'll need to know to use .NET Remoting in your real-world applications. I show you the differences between Singleton and SingleCall objects and untangle the mysteries of client-activated objects. I also introduce you to SoapSuds, which can be used to generate proxy objects containing only methods' stubs.

Chapter 4: Configuration and Deployment

This chapter introduces you to the aspects of configuration and deployment of .NET Remoting applications. It shows you how to use configuration files to avoid the hard coding of URLs or channel information for your remote object. You also learn about hosting your server-side components in Windows Services and IIS.

Chapter 5: Securing .NET Remoting

This chapter shows you how to leverage IIS's features when it comes to hosting your components in a secured environment. In this chapter, you learn how to enable basic HTTP sign-on and the more secure Windows-integrated authentication scheme, which is based on a challenge/response protocol. You also see how to enable encrypted access by using standard SSL certificates at the server side.

You will also read about ways to use .NET Remoting in a secure way when not relying on IIS.

Chapter 6: Creating Remoting Clients

Whenever I explain a new feature of the .NET Remoting framework, I tend to present it in an easily digestible console application to avoid having to show you numerous lines of boilerplate .NET code.

Of course, most of your real-world applications will either be Windows Forms or ASP.NET Web applications or Web Services. In this chapter, you therefore learn how to create remoting clients either as desktop or Web applications.

Chapter 7: In-Depth .NET Remoting

As a developer of distributed applications using .NET Remoting, you have to consider several fundamental differences from other remoting techniques and, of course, from the development of local applications. These differences, including lifetime management, versioning, and the handling of asynchronous calls and events, are covered in this chapter.

Chapter 8: The Ins and Outs of Versioning

Here you learn how to create .NET Remoting applications that are version resilient in a way that allows you to support different versions of clients with the same server.

Chapter 9: .NET Remoting Tips and Best Practices

In this chapter, I introduce you to a number of best practices that I've learned in more than three years of using .NET Remoting in numerous projects. This chapter will help you to increase scalability, performance, and stability of your distributed applications.

Chapter 10: Troubleshooting .NET Remoting

Unfortunately, things can and will go wrong at some point in time. That's why this chapter gives you a number of techniques and tools that help you to troubleshoot various issues you might encounter when using .NET Remoting. But don't be afraid: most of these can be remedied in a very brief amount of time.

Chapter 11: Inside the Framework

.NET provides an unprecedented extensibility for the remoting framework. The layered architecture of the .NET Remoting framework can be customized by either completely replacing the existing functionality of a given tier or chaining new implementation with the baseline .NET features.

Before working on the framework and its extensibility, I really encourage you to get a thorough understanding of the existing layers and their inner workings in this architecture. This chapter gives you that information.

Chapter 12: Creation of Sinks

This chapter covers the instantiation of message and channel sinks and sink chains. It shows you the foundation on which to build your own sinks—something you need to know before tackling the implementation of custom sinks.

Chapter 13: Extending .NET Remoting

This chapter builds on the information from Chapters 7 and 8 and shows you how to implement custom remoting sinks. This includes channel sinks that compress or encrypt the transported information, and message sinks to pass additional runtime information from a client to the server or to change the .NET Remoting programming model. This chapter concludes with showing you how to implement custom remoting proxies that forward method calls to remote objects.

Chapter 14: Developing a Transport Channel

This chapter builds on the information you gained in Chapters 7, 8, and 9 and presents the development of a custom .NET Remoting channel that transfers messages via standard Internet e-mail by using SMTP and POP3. It shows not only the implementation of this channel, but also the necessary phase of analyzing the underlying protocol to combine it with the features and requirements of .NET Remoting.

Chapter 15: Context Matters

This last chapter is about message-based processing in local applications. Here you learn how you can intercept calls to objects to route them through IMessageSinks. This routing allows you to create and maintain parts of your application's business logic at the metadata level by using custom attributes. You also discover why it might or might not be a good idea to do so.

Appendix A: .NET Remoting Usage Reference

This first appendix includes reference information you'll need when using .NET Remoting in your application. You'll learn about all the namespaces involved when creating clients and servers, and configuring and troubleshooting your application.

Appendix B: .NET Remoting Extensibility Reference

This second appendix covers the namespaces, classes, and interfaces that allow you to extend the .NET Remoting framework.

Appendix C: .NET Remoting Links

At the end of this book are collected a number of links to additional .NET Remoting–specific content on the Web. This includes everything from Microsoft-provided additional articles to custom channels and remoting extensions.

Source Code Download

You can find all source code presented in this book at the Apress download page at `http://www.apress.com`. If you have further suggestions or comments or want to access even more sample code on .NET Remoting, you are invited to visit thinktecture's .NET Remoting FAQ, which is hosted at `http://www.thinktecture.com/Resources/RemotingFAQ`.

We hope that you will benefit from the techniques and information we provide in this book when building your distributed applications based on the .NET Framework.

Ingo Rammer and Mario Szpuszta
Vienna, Austria

PART 1

Understanding

CHAPTER 1

Introduction to Remoting

This chapter gives you a short introduction to the world of distributed application development and its respective technologies. Here you get a chance to examine some scenarios in which .NET Remoting can be employed and learn some historical background on the progress and development of various remoting frameworks during the last ten years.

What Is Remoting?

Remoting is the process of programs or components interacting across certain boundaries. These contexts will normally resemble either different processes or machines.[1] In the .NET Framework, this technology provides the foundation for distributed applications—it simply replaces DCOM.

Remoting implementations generally distinguish between *remote objects* and *mobile objects*. The former provide the ability to execute methods on remote servers, passing parameters and receiving return values. The remote object will always "stay" at the server, and only a reference to it will be passed around among other machines.

When mobile objects pass a context boundary, they are serialized (marshaled) into a general representation—either a binary or a human readable format like XML—and then deserialized in the other context involved in the process. Server and client both hold copies of the same object. Methods executed on those copies of the object will always be carried out in the local context, and no message will travel back to the machine from which the object originated. In fact, after serialization and deserialization, the copied objects are indistinguishable from regular local objects, and there is also no distinction between a server object and a client object.

Scenarios for .NET Remoting

At the beginning of the client/server era, remoting was mostly used for accessing a server's resources. Every database or file server is an implementation of some technique that allows code to be executed remotely. Programming these older frameworks was so difficult a task that few products except for these server-side core services implemented remoting.

---

1. .NET extends this concept to include the ability to define additional contexts within one running application. Object accesses crossing these boundaries will pass the .NET Remoting framework as well.

Nowadays the building of distributed applications has gotten a lot easier so that it's quite feasible to distribute business applications among various machines to improve performance, scalability, and maintainability.

Centralized Business Logic

One of the key scenarios for implementing remoting is the concentration of business logic on one or more central servers. This considerably simplifies the maintainability and operability of large-scale applications. Changes in business logic do not entail your having to roll out an application to your organization's 10,000 worldwide users—you just have to update one single server.

When this centralized business logic is shared among different applications, this labor-saving effect multiplies considerably; instead of patching several applications, you just have to change the server's implementation.

Physical Separation of Layers

The security of a company's vital databases represents a common concern in this time of Web-enabled businesses. The general recommendation is against directly connecting from the Web server to the database because this setup would allow attackers easy access to critical data after they have seized control of the Web server.

Instead of this direct connection, an intermediate application server is introduced. This server is placed in a so-called demilitarized zone (DMZ), located between two firewalls. Firewall #1 only allows connections from the Web server to the app server, and Firewall #2 only allows connections from the app server to the databases.

Because the application server doesn't allow the execution of arbitrary SQL statements, yet provides object-oriented or function-based access to business logic, a security compromise of the Web server (which can only talk to the app server) is noncritical to a company's operations.

Accessing Other Platforms

In today's mid- to large-scale enterprises, you will normally encounter a heterogeneous combination of different platforms, frameworks, and programming languages. It is not uncommon to find that a bunch of tools have been implemented: Active Server Pages (ASP), Java Server Pages (JSP), PHP, or ColdFusion for Web applications, Visual Basic or Java for in-house applications, C++ for server-side batch jobs, scripting languages for customizing CRM systems, and so on.

Integrating these systems can be a daunting task for system architects. Remoting architectures like CORBA, SOAP, and .NET Remoting are an absolute necessity in large-scale enterprise application integration. (CORBA and SOAP are introduced and compared later in this chapter.)

Evolution of Remoting

The scenarios presented thus far have only been possible due to the constant evolution of remoting frameworks. The implementation of large-scale business applications in a distributed manner has only been practicable after the technical problems have been taken care of by the frameworks. CORBA, COM+, and EJB started this process several years ago, and .NET Remoting simplifies this process even more.

To underscore how far remoting has evolved from its cumbersome beginnings, the following sections give you a brief history of the various remoting frameworks.

DCE/RPC

Distributed Computing Environment (DCE), designed by the Open Software Foundation (OSF) during the early 1990s, was created to provide a collection of tools and services that would allow easier development and administration of distributed applications. The DCE framework provides several base services such as Remote Procedure Calls (DCE/RPC), Security Services, Time Services, and so on.

Implementing DCE is quite a daunting task; the interfaces have to be specified in Interface Definition Language (IDL) and compiled to C headers, client proxies, and server stubs by an IDL compiler. When implementing the server, one has to link the binary with DCE/Threads, which are available for C/C++. The use of programming languages other than these is somewhat restricted due to the dependence on the underlying services, like DCE/Threads, with the result that one has to live with single-threaded servers when refraining from using C/C++.

DCE/RPC nevertheless is the foundation for many current higher-level protocols including DCOM and COM+. Several application-level protocols such as MS SQL Server, Exchange Server, Server Message Block (SMB), which is used for file and printer sharing, and Network File System (NFS) are also based on DCE/RPC.

CORBA

Designed by the Object Management Group (OMG), an international consortium of about 800 companies, CORBA's aim is to be the middleware of choice for heterogeneous systems. OMG's CORBA, which stands for *Common Object Request Broker Architecture*, is only a collection of standards; the implementation of object request brokers (ORBs) is done by various third parties. Because parts of the standard are optional and the vendors of ORBs are allowed to include additional features that are not in the specifications, the world has ended up with some incompatible request brokers. As a result, an application developed to make use of one vendor's features could not easily be ported to another ORB. When you buy a CORBA-based program or component, you just can't be sure if it will integrate with your CORBA applications, which probably were developed for a different request broker.

Aside from this potential problem, CORBA also has quite a steep learning curve. The standard reads like a complete wish list of everything that's possible with remoted components—sometimes it simply is too much for the "standard business." You'll probably end up reading documents for days or weeks before your first request is ever sent to a server object.

Nevertheless, when you have managed to implement your first CORBA application, you'll be able to integrate a lot of programming languages and platforms. There are even layers for COM or EJB integration, and apart from SOAP, CORBA is the only true multiplatform, multi-programming language environment for distributed applications.

DCOM

Distributed Component Object Model (DCOM) is an "extension" that fits in the Component Object Model (COM) architecture, which is a binary interoperability standard that allows for component-oriented application development. You'll usually come in contact with COM when using ActiveX controls or ActiveX DLLs.

DCOM allows the distribution of those components among different computers. Scalability, manageability, and its use in WANs pose several issues that need to be addressed. DCOM uses

a pinging process to manage the object's lifetimes; all clients that use a certain object will send messages after certain intervals. When a server receives these messages, it knows that the client is still alive; otherwise it will destroy the object.

Additionally, reliance on the binary DCE/RPC protocol poses the need for direct TCP connections between the client and its server. Use of HTTP proxies is not possible. DCOM is available for Microsoft Windows and for some UNIX dialects (ported by the German company Software AG).

MTS/COM+

COM+, formerly Microsoft Transaction Server (MTS), was Microsoft's first serious attempt to reach into the enterprise application domain. It not only serves as a remoting platform, but also provides transaction, security, scalability, and deployment services. COM+ components can even be used via Microsoft Message Queue Server to provide asynchronous execution of methods.

Despite its advantages, COM+ does not yet support the automatic marshalling of objects to pass them by value between applications; instead you have to pass your data structures using ADO recordsets or other means of serialization. Other disadvantages that keep people from using COM+ are the somewhat difficult configuration and deployment, which complicates its use for real-world applications.

Java RMI

Traditional *Java Remote Method Invocation* (Java RMI) uses a manual proxy/stub compilation cycle. In contrast to DCE/RPC and DCOM, the interfaces are not written in an abstract IDL but in Java. This is possible due to Java being the only language for which the implementation of RMI is possible.

This limitation locked RMI out of the game of enterprise application integration. Even though all relevant platforms support a Java Virtual Machine, integration with legacy applications is not easily done.

Java EJB

Enterprise Java Beans (EJB) was Sun's answer to Microsoft's COM+. Unlike CORBA, which is only a standard, EJB comes with a reference implementation. This allows developers to check if their products run in any standard-complying EJB container. EJB has been widely accepted by the industry, and there are several container implementations ranging from free open source to commercial implementations by well-known middleware vendors.

One problem with EJB is that even though a reference implementation exists, most vendors add features to their application servers. When a developer writes a component that uses one of those features, the application will not run on another vendor's EJB container.

Former versions of EJB have been limited to the Java platform because of their internal reliance on RMI. The current version allows the use of IIOP, which is the same transfer protocol CORBA uses, and third parties already provide commercial COM/EJB bridges.

Web Services/SOAP/XML-RPC

Web Services provided the first easy to understand and implement solution to true cross-platform and cross-language interoperability. Web Services technically are stateless calls to remote components via HTTP POST with a payload encoded in some XML format.

Two different XML encodings are currently in major use: XML-RPC and SOAP. *XML-RPC* can be described as a poor man's SOAP. It defines a very lightweight protocol with a specification size of about five printed pages. Implementations are already available for a lot of programming environments, ranging from AppleScript to C/C++, COM, Java, Perl, PHP, Python, Tcl, and Zope—and of course there's also an implementation for .NET.

SOAP, or *Simple Object Access Protocol*, defines a much richer set of services; the specification covers not only remote procedure calls, but also the *Web Services Description Language* (WSDL) and *Universal Description, Discovery, and Integration* (UDDI). WSDL is SOAP's interface definition language, and UDDI serves as a directory service for the discovery of Web Services. Those additional protocols and specifications are also based on XML, which allows all SOAP features to be implemented on a lot of platforms.

The specifications and white papers for SOAP, WSDL, UDDI, and corresponding technologies cover several hundred pages, and you can safely assume that this document will grow further when topics like routing and transactions are addressed. Fortunately for .NET developers, the .NET platform takes care of *all* issues regarding SOAP.

.NET Remoting

At first look, .NET Remoting is to Web Services what ASP has been to CGI programming. It takes care of a lot of issues for you: contrary to Web Services, for example, .NET Remoting enables you to work with stateful objects.

In addition to the management of stateful objects, .NET Remoting gives you a flexible and extensible framework that allows for different transfer mechanisms (HTTP and TCP are supported by default), encodings (SOAP and binary come with the framework), and security settings (IIS Security and SSL come out of the box).

With these options, and the possibility of extending all of them or providing completely new implementations, .NET Remoting is well suited to today's distributed applications. You can choose between HTTP-based transport for the Internet or a faster TCP-based one for LAN applications by literally changing a single line in a configuration file.

Interface description does not have to be manually coded in any way, even though it's supported if you like to design your applications this way. Instead, metadata can be extracted from running servers, or from any .NET assembly.

Summary

This chapter provided a short introduction to the world of distributed application development and the respective technologies. You now know about the various scenarios in which .NET Remoting can be applied and understand how it differs from other distributed application protocols and techniques.

CHAPTER 2

.NET Remoting Basics

This chapter gets you started with your first remoting application. Before going directly into the code, I present the differences between .NET Remoting and other distributed application frameworks. I then introduce you to the basic types of remote objects, server-activated objects, and client-activated objects, and show you how to pass data by value. I also give you some basic information about lifetime management issues and the generation of metadata, which is needed for the client to know about the interfaces of the server-side objects.

Advantages of .NET Remoting

As you've seen in the previous chapter, several different architectures for the development of distributed applications already exist. You might therefore wonder why .NET introduces another, quite different way of developing those kinds of applications. One of the major benefits of .NET Remoting is that it's centralized around well-known and well-defined standards like HTTP and that it is directly tied to the .NET Framework and has not been retrofitted later.

Ease of Implementation

Comparing .NET Remoting to other remoting schemas is like comparing COM development in Visual Basic to C++. Visual Basic 6 allowed developers to concentrate on the business needs their applications had to fulfill without having to bother with the technical details of COM. The C++ programmers had to know the exact specifications of COM (at least before the introduction of ATL) and implement truckloads of code for supporting them.

With .NET this concept of absolute ease of implementation has been extended to the development of distributed applications. There are no proxy/stub-compilation cycles as in Java RMI. You don't have to define your interfaces in a different programming language as you would with CORBA or DCOM. A unique feature is that you don't have to decide up front on the encoding format of remoting requests; instead, you can switch from a fast TCP transport to HTTP by changing one word in a configuration file. You can even provide both communication channels for the same objects by adding another line to the configuration. You are not fixed on one platform or programming language as with DCOM, COM+, and Java EJB. Configuration and deployment is a lot easier than it was in DCOM.

Even though .NET Remoting provides a lot of features, it doesn't lock you in. Quite the contrary: it can be as easy as you like or as complex as you need. The process of enabling remoting for an object can be as straightforward as writing two lines of code or as sophisticated as implementing a given transfer protocol or format on your own.

Extensible Architecture

.NET Remoting offers the developer and administrator a vastly greater choice of protocols and formats than any of the former remoting mechanisms. In Figure 2-1, you can see a simplified view of the .NET Remoting architecture. Whenever a client application holds a reference to a remote object, it will be represented by a TransparentProxy object, which "masquerades" as the destination object. This proxy will allow all of the target object's instance methods to be called upon it. Whenever a method call is placed to the proxy, it will be converted into a message, and the message will pass various layers.

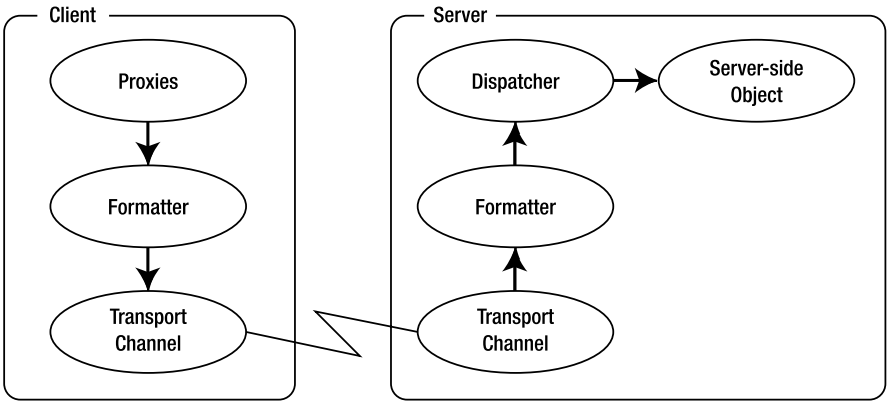

Figure 2-1. *The .NET Remoting architecture (simplified)*

The message will pass a serialization layer—the formatter—which converts it into a specific transfer format such as SOAP. The serialized message later reaches a transport channel, which transfers it to a remote process via a specific protocol like HTTP or TCP. On the server side, the message also passes a formatting layer, which converts the serialized format back into the original message and forwards it to the dispatcher. Finally, the dispatcher calls the target object's method and passes back the response values through all tiers. This architecture is shown in detail in Chapter 11.

In contrast to other remoting architectures, most layers can either be extended or completely replaced, and additional layers can be chained to the baseline .NET Remoting Framework to allow for custom processing of messages. (More about this in Chapters 11, 12, and 13.)

You can easily switch between implementations of the different layers without changing any source code. A remoting application that's been written using a binary TCP-based protocol can be opened for third parties using a SOAP/HTTP-based protocol by changing some lines in a configuration file to replace the .NET Remoting transport channel.

Interface Definitions

Most remoting systems like DCE/RPC, RMI, and J2EE demand a manual creation of so-called proxy/stub objects. The proxy encapsulates the connection to the remote object on the client and forwards calls to a stub object on the server, which in turn passes them on to the "real" object. In most of these environments (at least in CORBA, DCE/RPC, and DCOM) the "source code" for generating these objects has to be written in an abstract Interface Definition Language and precompiled to generate implementation headers for a certain programming language.

In comparison to this traditional approach, .NET Remoting uses a generic proxy for all kinds of remote objects. This is possible because .NET is the first framework that has been designed with remoting in mind; on other platforms these capabilities have been retrofitted and therefore have to be integrated into the given architecture and programming model.

Such ease of remoting poses the potential problem of your using an incorrect design.[1] This book will help you to make the right architectural decisions. For example, even though you don't have to write any interface definitions in IDL, you still should separate interface from implementation; you can, however, write both in the same language—in *any* .NET programming language.

.NET Remoting provides several different ways of defining those interfaces, as discussed in the following sections.

Shared Assembly

In this case, the server-side object's implementation exists on the client as well. Only during instantiation is it determined whether a local object or an object on the remote server will be created. This method allows for a semitransparent switch between invoking the local implementation (for example, when working offline) and invoking server-side objects (for example, to make calculations on better-performing servers when connected to the network).

When using this method with "conventional" distributed applications that don't need to work in a disconnected scenario, you need to use a lot of care, because it poses some risks due to easy-to-miss programming and configuration errors. When the object is mistakenly instantiated as a local object on the client and passed to the server (as a method's parameter, for example) you might run into serious troubles, ranging from InvalidCastExceptions to code that works in the development environment but doesn't work in the production environment because of firewall restrictions. In this case the client has in reality become the server, and further calls to the object will pass from the server to your clients.

Shared Interfaces or Base Objects

When creating a distributed application, you define the base classes or interfaces to your remote objects in a separated assembly. This assembly is used on both the client and the server. The real implementation is placed only on the server and is a class that extends the base class or implements the interface.

The advantage is that you have a distinct boundary between the server and the client application, but you have to build this intermediate assembly as well. Good object-oriented practices nevertheless recommend this approach!

---

1. This is partly the same as it was in Visual Basic 6. VB 6 allowed you to create applications without a lot of up-front design work. This often led to applications that were hardly maintainable in the long run.

■ **Note** This is the recommended way of creating .NET Remoting applications.

Generated Metadata Assembly

This approach seems to be the most elegant one at first glance. You develop the server in the same way as when using the shared assemblies method. Instead of really sharing the DLL or EXE, you later extract the necessary metadata, which contains the interface information, using SoapSuds.

SoapSuds will either need the URL to a running server or the name of an assembly as a parameter, and will extract the necessary information (interfaces, base classes, objects passed by value, and so on). It will put this data into a new assembly, which can be referenced from the client application. You can then continue to work as if you'd separated your interfaces right from the beginning.

■ **Caution** Even though using SoapSuds might seem intriguing when you look at it for the first time, experience shows otherwise. Nowadays, Microsoft recommends using this tool in only very specific cases as detailed in Chapter 9.

Serialization of Data

With the exception of earlier TCP/IP RPC implementations, in which you even had to worry about little-endian/big-endian conversions, all current remoting frameworks support the automatic encoding of simple data types into the chosen transfer format. The problem starts when you want to pass a copy of an object from server to client. Java RMI and EJB support these requirements, but COM+, for example, did not. The commonly used serializable objects within COM+ were PropertyBag and ADO Recordsets—but there was no easy way of passing large object structures around.

In .NET Remoting the encoding/decoding of objects is natively supported. You just need to mark such objects with the [Serializable] attribute or implement the interface ISerializable and the rest will be taken care of by the framework.

The underlying .NET runtime formatting mechanism marshals simple data types and subobjects (which have to be serializable or exist as remote objects), and even ensures that circular references will be tracked and transferred correctly.

Lifetime Management

In distributed applications there are generally three ways of managing lifetime. The first is to have an open network connection (for example, using TCP) from the client to the server. Whenever this connection is terminated, the server's memory will be freed.

Another possibility is the DCOM approach, where a combined reference counting and pinging mechanism is used. In this case the server receives messages from its clients at certain intervals. As soon as no more messages are received, it will free its resources.

In the Internet age, in which you don't know your users up front, you cannot rely on the possibility of creating a direct TCP connection between the client and the server. Your users might be sitting behind a firewall that only allows HTTP traffic to pass through. The same router will block any pings the server might send to your users. Because of those issues, the .NET Remoting lifetime service is customizable as well. By default an object will get a lifetime assigned to it, and each call from the client will reset this "time to live." The .NET Framework also allows a so-called sponsor to be registered with a server-side object. It will be contacted just before the lifetime is over and can also increase the object's time to live.

The combination of these two approaches allows for a configurable lifetime service that does not depend on any specific connection from the server to the client.

■ **Note** This is one of the core features of .NET Remoting: it never depends on any existing connections. These are created and destroyed on demand.

Multiserver/Multiclient

When you use remote objects (as opposed to using copies of remotely generated objects that are passed by value), .NET automatically keeps track of where they originated. So a client can ask one server to create an object and safely pass this as a method's parameter to another server.

The second server will then directly execute its methods on the first server, without a round-trip through the client. Nevertheless, this also means there has to be a direct way of communication from the second server to the first one—that is, there must not be a firewall in between, or at least the necessary ports should be opened.

Your First Remoting Application

In the following sections, you create a sample .NET Remoting application that demonstrates some of the concepts discussed earlier in this chapter. First and foremost, there are two very different kinds of objects when it comes to remoting: objects that are passed by reference and those that are passed by value. *MarshalByRefObjects*[2] allow you to execute remote method calls on the server side. These objects will live on the server and only a so-called ObjRef will be passed around. You can think of the ObjRef as a networked pointer that shows on which server the object lives and contains an ID to uniquely identify the object. The client will usually not have the compiled objects in one of its assemblies; instead only an interface or a base class will be available. Every method, including property gets/sets, will be executed on the server. The .NET Framework's proxy objects will take care of all remoting tasks, so that the object will look just like a local one on the client.

The second kind of objects will be referred to as *ByValue objects* or *serializable objects* throughout this book. When these objects are passed over remoting boundaries (as method parameters or return values), they are serialized into a string or a binary representation and restored as a copy on the other side of the communications channel. After this re-creation, there is no notation of client or server for this kind of object; each one has its own copy, and

---

2. Called so because every object of this kind has to extend System.MarshalByRefObject, or one of its children.

both run absolutely independently. Methods called on these objects will execute in the same context as the origination of the method call. For example, when the client calls a function on the server that returns a ByValue object, the object's state (its instance variables) will be transferred to the client and subsequent calls of methods will be executed directly on the client. This also means that the client has to have the compiled object in one of its assemblies. The only other requirement for an object to be passable by value is that it supports serialization. This is implemented using a class-level attribute: *[Serializable]*. In addition to this "standard" serialization method, you'll also be able to implement ISerializable, which I show you how to do in Chapter 6.

The First Sample

This sample remoting application exposes a server-side MarshalByRefObject in Singleton mode. You will call this object *CustomerManager*, and it will provide a method to load a *Customer* object (which is a ByValue object) from a fictitious database. The resulting object will then be passed as a copy to the client.

Architecture

When using remote objects, both client and server must have access to *the same* interface definitions and serializable objects that are passed by value. This leads to the general requirement that at least three assemblies are needed for any .NET Remoting project: a shared assembly, which contains serializable objects and interfaces or base classes to MarshalByRefObjects; a server assembly, which implements the MarshalByRefObjects; and a client assembly, which consumes them.

■ **Note** It is not sufficient to copy and paste interface definitions from the server's source code directly into the client's. Instead, they really have to share a reference to the same DLL because the assembly's name becomes part of the complete type name. The interface ICustomerManager in the assembly server.exe would therefore be completely independent from (and different from) the interface ICustomerManager in the assembly client.exe.

In most of the examples throughout this book, you will end up with these three assemblies:

- *General*: This represents the shared assembly, which contains the interface ICustomerManager and the ByValue object Customer. As the methods of a Customer object will be executed either on the client or on the server, its implementation is contained within the General assembly as well.
- *Server*: This assembly contains the server-side implementation of CustomerManager.
- *Client*: This assembly contains a sample client.

Defining the Remote Interface

As a first step, you have to define the interface ICustomerManager, which will be implemented by the server. In this interface, you'll define a single method, `GetCustomer()`, that returns a Customer object to the caller.

```
public interface ICustomerManager
{
   Customer GetCustomer(int id);
}
```

This interface will allow the client to load a Customer object by a given ID.

Defining the Data Object

Because you want to provide access to customer data, you first need to create a Customer class that will hold this information. This object needs to be passed as a copy (by value), so you have to mark it with the [Serializable] attribute.

In addition to the three properties `FirstName`, `LastName`, and `DateOfBirth`, you will also add a method called `GetAge()` that will calculate a customer's age. Next to performing this calculation, this method will write a message to the console so that you can easily see in which context (client or server) the method is executing.

```
[Serializable]
public class Customer
{
   public String FirstName;
   public String LastName;
   public DateTime DateOfBirth;

   public Customer()
   {
      Console.WriteLine(Customer.constructor: Object created);
   }

   public int GetAge()
   {
      Console.WriteLine("Customer.GetAge(): Calculating age of {0}, " +
         "born on {1}.",
         FirstName,
         DateOfBirth.ToShortDateString());

      TimeSpan tmp = DateTime.Today.Subtract(DateOfBirth);
      return tmp.Days / 365; // rough estimation
   }
}
```

Up to this point in the code, there's not much difference from a local application. Before being able to start developing the server, you have to put the interface and the class in the namespace General and compile this project to a separate DLL, which will later be referenced by the server and the client.

Implementing the Server

On the server you need to provide an implementation of ICustomerManager that will allow you to load a customer from a fictitious database; in the current example, this implementation will only fill the Customer object with static data.

■ **Note** To create concise samples, I will present throughout the book mostly console applications that focus on demonstrating one single aspect of .NET Remoting at a time. Using console applications as servers, however, is not recommended in production environments, and I'll discuss the more serious hosting options (Windows services and Internet Information Server) in Chapter 4. It's important to note that you can still use all the demonstrated techniques no matter which host you choose—it's just that console applications are easier to use as a tool to explain concepts.

To implement the sample server, you create a new console application in Visual Studio .NET called Server and add a reference to the framework assembly System.Runtime.Remoting.dll and the newly compiled General.dll from the previous step (you will have to use the Browse button here, because you didn't copy the assembly to the global assembly cache [GAC]). The server will have to access the namespace General and System.Runtime.Remoting plus a remoting channel, so you have to add the following lines to the declaration:

```
using System.Runtime.Remoting;
using General;
using System.Runtime.Remoting.Channels;
using System.Runtime.Remoting.Channels.Http;
```

As described previously, you will have to implement ICustomerManager in an object derived from MarshalByRefObject. The method `GetCustomer()`will just return a dummy Customer object:

```
class CustomerManager: MarshalByRefObject, ICustomerManager
{
   public CustomerManager()
   {
      Console.WriteLine("CustomerManager.constructor: Object created");
   }

   public Customer GetCustomer(int id)
   {
      Console.WriteLine("CustomerManager.GetCustomer(): Called");
      Customer tmp = new Customer();
      tmp.FirstName = "John";
      tmp.LastName = "Doe";
      tmp.DateOfBirth = new DateTime(1970,7,4);
      Console.WriteLine("CustomerManager.GetCustomer(): Returning " +
                        "Customer-Object");
      return tmp;
   }
}
```

It still looks more or less the same as a "conventional" nonremoting class would—the only difference is that the class doesn't inherit directly from System.Object, but from System.MarshalByRefObject.

Now let's have a look at the server startup code. This is a very basic variant of registering a server-side object. It doesn't yet use a configuration file, but the server's parameters are hard coded in `void Main()`.

```
class ServerStartup
{
   static void Main(string[] args)
   {
      HttpChannel chnl = new HttpChannel(1234);
      ChannelServices.RegisterChannel(chnl);
      RemotingConfiguration.RegisterWellKnownServiceType(
                                 typeof(CustomerManager),
                                 "CustomerManager.soap",
                                 WellKnownObjectMode.Singleton);

      // the server will keep running until keypress.
      Console.ReadLine();
   }
}
```

Now take a closer look at the startup sequence of the server:

```
HttpChannel chnl = new HttpChannel(1234);
```

A new HTTP channel (System.Runtime.Remoting.Channels.Http.HttpChannel) is created and configured to listen on port 1234. The default transfer format for HTTP is SOAP.

```
ChannelServices.RegisterChannel(chnl);
```

The channel is registered in the remoting system. This will allow incoming requests to be forwarded to the corresponding objects.

```
RemotingConfiguration.RegisterWellKnownServiceType(
       typeof(CustomerManager),
       "CustomerManager.soap",
       WellKnownObjectMode.Singleton);
```

The class CustomerManager is registered as a WellKnownServiceType, which allows the client to remotely call its methods. The URL will be CustomerManager.soap—whereas this can be any string you like, the extension .soap or .rem should be used for consistency. This is absolutely necessary when hosting the components in IIS as it maps these two extensions to the .NET Remoting Framework (as shown in Chapter 4).

The object's mode is set to Singleton to ensure that only one instance will exist at any given time.

```
Console.ReadLine();
```

This last line is not directly a part of the startup sequence but just prevents the program from exiting while the server is running. You can now compile and start this server.

■ **Note** If you look closely at the startup sequence, you'll notice that the registered class is not directly bound to the channel. In fact, you'd be right in thinking that all available channels can be used to access all registered objects.

Implementing the Client

The sample client will connect to the server and ask for a Customer object. For the client you also need to add a reference to System.Runtime.Remoting.dll and the compiled General.dll from the preceding step (you will again have to use the Browse button, because you didn't copy the assembly to the GAC).

■ **Note** The same disclaimer as for the server application applies here. I will use console applications throughout the book because they allow me to demonstrate a single aspect of .NET Remoting at a time without cluttering the application code. All the techniques will also work for Windows Forms applications, Windows services, and ASP.NET applications. In Chapter 5, I'll show you how to create these other kinds of client applications.

The same using statements are needed as for the server:

```
using System.Runtime.Remoting;
using General;
using System;
using System.Runtime.Remoting.Channels.Http;
using System.Runtime.Remoting.Channels;
```

The `void Main()` method will register a channel, contact the server to acquire a Customer object, and print a customer's age.

```
class Client
{
   static void Main(string[] args)
   {
      HttpChannel channel = new HttpChannel();
      ChannelServices.RegisterChannel(channel);

      ICustomerManager mgr = (ICustomerManager) Activator.GetObject(
         typeof(ICustomerManager),
         "http://localhost:1234/CustomerManager.soap");
      Console.WriteLine("Client.Main(): Reference to CustomerManager acquired");

      Customer cust = mgr.GetCustomer(4711);
      int age = cust.GetAge();
      Console.WriteLine("Client.Main(): Customer {0} {1} is {2} years old.",
         cust.FirstName,
         cust.LastName,
         age);

      Console.ReadLine();
   }
}
```

Now let's take a detailed look at the client:

```
HttpChannel channel = new HttpChannel();
ChannelServices.RegisterChannel(channel);
```

With these two lines, the HTTP channel is registered on the client. It is not necessary to specify a port number here, because the client-side TCP port will be assigned automatically.

```
ICustomerManager mgr = (ICustomerManager) Activator.GetObject(
        typeof(ICustomerManager),
        "http://localhost:1234/CustomerManager.soap");
```

This line creates a local proxy object that will support the interface ICustomerManager.

Let's examine the call to Activator.GetObject() a little closer:

```
Activator.GetObject(typeof(ICustomerManager),
                    "http://localhost:1234/CustomerManager.soap");
```

Instead of using the `new` operator, you have to let the *Activator* create an object. You need to specify the class or interface of the object—in this case, ICustomerManager—and the URL to the server. This is not necessary when using configuration files—as shown in Chapter 4—because in that situation the `new` operator will know which classes will be remotely instantiated and will show the corresponding behavior.

In this example, the Activator will create a proxy object on the client side but will not yet contact the server.

```
Customer cust = mgr.GetCustomer(4711);
```

The `GetCustomer()` method is executed on the TransparentProxy[3] object. Now the first connection to the server is made and a message is transferred that will trigger the execution of `GetCustomer()` on the server-side Singleton object CustomerManager. You can verify this because you included a `Console.WriteLine()` statement in the server's `GetCustomer()` code. This line will be written into the server's console window.

The server now creates a Customer object and fills it with data. When the method returns, this object will be serialized and all public and private properties converted to an XML fragment. This XML document is encapsulated in a SOAP return message and transferred back to the client. The .NET Remoting Framework on the client now implicitly generates a new Customer object on the client and fills it with the serialized data that has been received from the server.

The client now has an exact copy of the Customer object that has been created on the server; there is *no* difference between a normal locally generated object and this serialized and deserialized one. All methods will be executed directly in the client's context! This can easily be seen in Figure 2-2, which shows the included `WriteLine()` statement in the Customer object's `GetAge()` method that will be output to the client's console window. Figure 2-3 shows the corresponding output of the server application.

---

3. This is the proxy object that has been returned from the call to `Activator.GetObject()`.

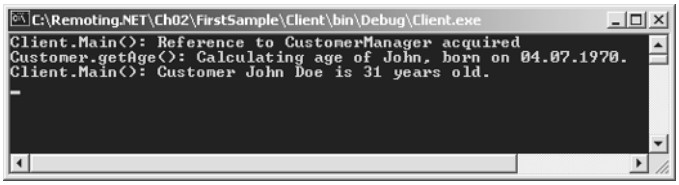
C:\Remoting.NET\Ch02\FirstSample\Client\bin\Debug\Client.exe
Client.Main(): Reference to CustomerManager acquired
Customer.getAge(): Calculating age of John, born on 04.07.1970.
Client.Main(): Customer John Doe is 31 years old.

Figure 2-2. *Client output for first sample*

Figure 2-3. *Server output for first sample*

Extending the Sample

Quite commonly, data has to be validated against several business rules. It's very convenient and maintainable to place this validation code on a central server. To allow validation of Customer data, you will extend the ICustomerManager interface to include a `validate()` method. This method will take a Customer object as a parameter and return another object by value. This returned object contains the status of the validation and explanatory text. As a sample business rule, you will check if the customer has been assigned a first name and last name and is between 0 and 120 years old.

General Assembly

In the General assembly extend the interface ICustomerManager to include the method `Validate()`.

```
public interface ICustomerManager
{
   Customer GetCustomer(int id);
   ValidationResult Validate (Customer cust);
}
```

The ValidationResult is defined as follows. It will be a serializable (transfer by value) object with a constructor to set the necessary values.

```
[Serializable]
public class ValidationResult
{
   public ValidationResult (bool ok, String msg)
   {
```

```
      Console.WriteLine("ValidationResult.ctor: Object created");
      this.Ok = ok;
      this.ValidationMessage = msg;
   }

   public bool Ok;
   public String ValidationMessage;
}
```

Server

On the server, you have to provide an implementation of the mentioned business rule:

```
public ValidationResult Validate(Customer cust)
{
   int age = cust.GetAge();
   Console.WriteLine("CustomerManager.Validate() for {0} aged {1}",
                     cust.FirstName, age);
   if ((cust.FirstName == null) || (cust.FirstName.Length == 0))
   {
      return new ValidationResult(false,"Firstname missing");
   }

   if ((cust.LastName == null) || (cust.LastName.Length == 0))
   {
      return new ValidationResult(false, "Lastname missing");
   }

   if (age < 0 || age > 120)
   {
      return new ValidationResult(false,"Customer must be " +
      "younger than 120 years");
   }

   return new ValidationResult(true,"Validation succeeded");
}
```

This function just checks the given criteria and returns a corresponding ValidationResult object, which contains the state of the validation (success/failure) and some explanatory text.

Client

To run this sample, you also have to change the client to create a new Customer object and let the server validate it.

```
static void Main(string[] args)
{
   HttpChannel channel = new HttpChannel();
   ChannelServices.RegisterChannel(channel);
```

```
   ICustomerManager mgr = (ICustomerManager) Activator.GetObject(
      typeof(ICustomerManager),
      "http://localhost:1234/CustomerManager.soap");
   Console.WriteLine("Client.main(): Reference to rem. object acquired");

   Console.WriteLine("Client.main(): Creating customer");
   Customer cust = new Customer();
   cust.FirstName = "Joe";
   cust.LastName = "Smith";
   cust.DateOfBirth = new DateTime(1800,5,12);

   Console.WriteLine("Client.main(): Will call validate");
   ValidationResult res = mgr.validate (cust);
   Console.WriteLine("Client.main(): Validation finished");
   Console.WriteLine("Validation result for {0} {1}\n-> {2}: {3}",
                     cust.FirstName, cust.LastName,res.Ok.ToString(),
                     res.ValidationMessage);

   Console.ReadLine();
}
```

As you can see in Figure 2-4, the Customer object is created in the client's context and then passed to the server as a parameter of `Validate()`. Behind the scenes the same thing happens as when `GetCustomer()` is called in the previous example: the Customer object will be serialized and transferred to the server, which will in turn create an exact copy.

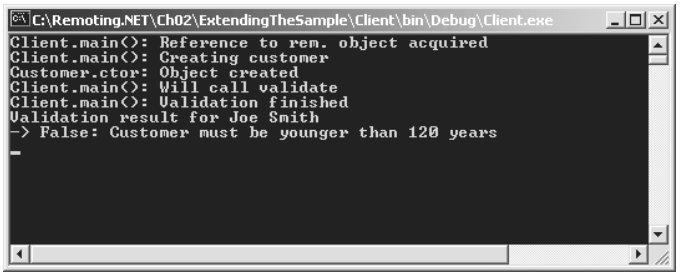
C:\Remoting.NET\Ch02\ExtendingTheSample\Client\bin\Debug\Client.exe
Client.main(): Reference to rem. object acquired
Client.main(): Creating customer
Customer.ctor: Object created
Client.main(): Will call validate
Client.main(): Validation finished
Validation result for Joe Smith
-> False: Customer must be younger than 120 years

Figure 2-4. *Client's output when validating a customer*

This copied object is used for validation against the defined business rules. When looking at the server's output in Figure 2-5, you will see that `CustomerManager.Validate()` and `Customer.GetAge()` are executed on the server. The returned ValidationResult is serialized and transferred to the client.

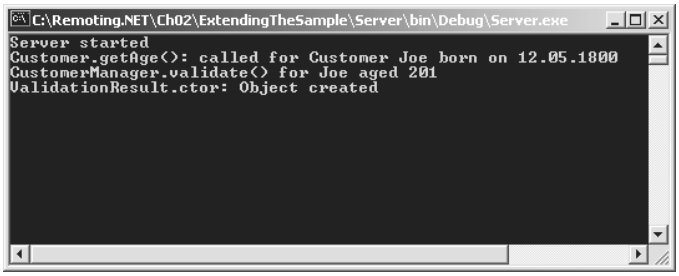

Figure 2-5. *Server's output while validating a customer*

Summary

In this chapter, you read about the basics of .NET Remoting. You now know the difference between MarshalByRefObjects, which allow you to call server-side methods, and ByValue objects, which have to be serializable and will be passed as copies. You read about the general structure of a remoting application and implemented a sample application that relied on shared interfaces.

CHAPTER 3

.NET Remoting in Action

In this chapter, I demonstrate the key techniques you'll need to know to use .NET Remoting in your real-world applications. I show you the differences between Singleton and SingleCall objects and untangle the mysteries of client-activated objects. I also introduce you to the different techniques to create the necessary metadata for your client applications. This chapter is somewhat code based, so prepare yourself to start VS .NET quite often!

Types of Remoting

As you have seen in the previous chapter's examples, there are two very different types of remote interaction between components. One uses serializable objects that are passed as a copy to the remote process. The second employs server-side (remote) objects that allow the client to call their methods.

ByValue Objects

Marshalling objects by value means to serialize their state (instance variables), including all objects referenced by instance variables, to some persistent form from which they can be deserialized in a different context. This ability to serialize objects is provided by the .NET Framework when you set the attribute [Serializable] for a class or implement ISerializable.

When passing the Customer object in the previous chapter's validation example to the server, it is serialized to XML like this:

```
<a1:Customer id="ref-4">
<FirstName id="ref-5">Joe</FirstName>
<LastName id="ref-6">Smith</LastName>
<DateOfBirth>1800-05-12T00:00:00.0000+02:00</DateOfBirth>
</a1:Customer>
```

This XML document is read by the server and an exact copy of the object is created.

Note An important point to know about ByValue objects is that **they are not *remote* objects**. All methods on those objects will be executed locally (in the same context) to the caller.. This also means that, unlike with MarshalByRefObjects, the compiled class has to be available to the client. You can see this in the preceding snippet, where "age" is not serialized but will be recalculated at the client using the `GetAge()` method.

When a ByValue object holds references to other objects, those have to be either serializable or MarshalByRefObjects; otherwise, an exception will be thrown, indicating that those objects are not remoteable.

MarshalByRefObjects

A MarshalByRefObject is a remote object that runs on the server and accepts method calls from the client. Its data is stored in the server's memory and its methods executed in the server's AppDomain. Instead of passing around a variable that points to an object of this type, in reality only a pointer-like construct—called an ObjRef—is passed around. Contrary to common pointers, this ObjRef does not contain the memory address, rather the server name/IP address and an object identity that identifies exactly *one* object of the many that are probably running on the server. I cover this in depth later in this chapter. MarshalByRefObjects can be categorized into two groups: server-activated objects (SAOs) and client-activated objects (CAOs).

Server-Activated Objects

Server-activated objects are somewhat comparable to classic stateless Web Services. When a client requests a reference to a SAO, no message will travel to the server. Only when methods are called on this remote reference will the server be notified.

Depending on the configuration of its objects, the server then decides whether a new instance will be created or an existing object will be reused. SAOs can be marked as either Singleton or SingleCall. In the first case, one instance serves the requests of all clients in a multithreaded fashion. When using objects in SingleCall mode, as the name implies, a new object will be created for each request and destroyed afterwards.

■ **Note** As Singleton objects will be accessed by multiple threads at the same time, it's important that you use correct locking and resource sharing patterns to prevent data corruption. For example, if your objects access a database, you should never use a class-level member that holds a SqlConnection object. Instead, you should create and destroy this connection object directly inside the methods.

In the following examples, you'll see the differences between these two kinds of services. You'll use the same shared interface, client- and server-side implementation of the service, and only change the object mode on the server.

The shared assembly *General.*dll will contain the interface to a very simple remote object that allows the storage and retrieval of stateful information in the form of an int value, as shown in Listing 3-1.

Listing 3-1. *The Interface Definition That Will Be Compiled to a DLL*

```
using System;

namespace General
{
   public interface IMyRemoteObject
   {
```

```
      void SetValue (int newval);
      int GetValue();
   }
}
```

The client that is shown in Listing 3-2 provides the means for opening a connection to the server and tries to set and retrieve the instance values of the server-side remote object. You'll have to add a reference to System.Runtime.Remoting.DLL to your Visual Studio .NET project for this example.

Listing 3-2. *A Simple Client Application*

```
using System;
using System.Runtime.Remoting;
using General;
using System.Runtime.Remoting.Channels.Http;
using System.Runtime.Remoting.Channels;

namespace Client
{

   class Client
   {
      static void Main(string[] args)
      {
         HttpChannel channel = new HttpChannel();
         ChannelServices.RegisterChannel(channel);

         IMyRemoteObject obj = (IMyRemoteObject) Activator.GetObject(
            typeof(IMyRemoteObject),
            "http://localhost:1234/MyRemoteObject.soap");
         Console.WriteLine("Client.Main(): Reference to rem. obj acquired");

         int tmp = obj.GetValue();
         Console.WriteLine("Client.Main(): Original server side value: {0}",tmp);

         Console.WriteLine("Client.Main(): Will set value to 42");
         obj.SetValue(42);

         tmp = obj.GetValue();
         Console.WriteLine("Client.Main(): New server side value {0}", tmp);

         Console.ReadLine();
      }
   }
}
```

The sample client will read and output the server's original value, change it to 42, and then read and output it again.

SingleCall Objects

For SingleCall objects the server will create a single object, execute the method, and destroy the object again. SingleCall objects are registered at the server using the following statement:

```
RemotingConfiguration.RegisterWellKnownServiceType(
        typeof(<YourClass>), "<URL>",
        WellKnownObjectMode.SingleCall);
```

Objects of this kind can obviously not hold any state information, as all internal variables will be discarded at the end of the method call. The reason for using objects of this kind is that they can be deployed in a very scalable manner. These objects can be located on different computers with an intermediate multiplexing/load-balancing device, which would not be possible when using stateful objects. The complete server for this example can be seen in Listing 3-3.

Listing 3-3. *The Complete Server Implementation*

```
using System;
using System.Runtime.Remoting;
using General;
using System.Runtime.Remoting.Channels.Http;
using System.Runtime.Remoting.Channels;

namespace Server
{

   class MyRemoteObject: MarshalByRefObject, IMyRemoteObject
   {
      int myvalue;

      public MyRemoteObject()
      {
         Console.WriteLine("MyRemoteObject.Constructor: New Object created");
      }

      public MyRemoteObject(int startvalue)
      {
         Console.WriteLine("MyRemoteObject.Constructor: .ctor called with {0}",
                           startvalue);
         myvalue = startvalue;
      }

      public void SetValue(int newval)
      {
         Console.WriteLine("MyRemoteObject.setValue(): old {0} new {1}",
                           myvalue,newval);
         myvalue = newval;
      }
```

```
        public int GetValue()
        {
            Console.WriteLine("MyRemoteObject.getValue(): current {0}",myvalue);
            return myvalue;
        }
    }

    class ServerStartup
    {
        static void Main(string[] args)
        {
            Console.WriteLine ("ServerStartup.Main(): Server started");

            HttpChannel chnl = new HttpChannel(1234);
            ChannelServices.RegisterChannel(chnl);
            RemotingConfiguration.RegisterWellKnownServiceType(
                  typeof(MyRemoteObject),
                  "MyRemoteObject.soap",
                  WellKnownObjectMode.SingleCall);

            // the server will keep running until keypress.
            Console.ReadLine();
        }
    }
}
```

When the program is run, the output in Figure 3-1 will appear on the client.

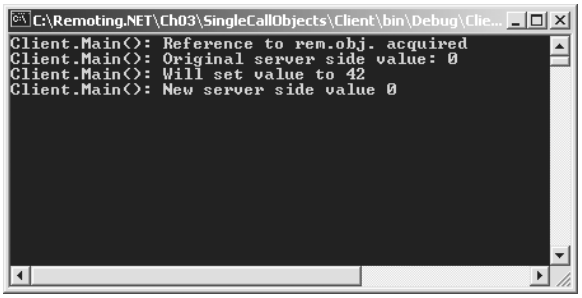
C:\Remoting.NET\Ch03\SingleCallObjects\Client\bin\Debug\Clie...
Client.Main(): Reference to rem.obj. acquired
Client.Main(): Original server side value: 0
Client.Main(): Will set value to 42
Client.Main(): New server side value 0

Figure 3-1. *Client's output for a SingleCall object*

What's happening is exactly what you'd expect from the previous description—even though it might not be what you'd normally expect from an object-oriented application. The reason for the server returning a value of 0 after setting the value to 42 is that your client is talking to a completely different object. Figure 3-2 shows the server's output.

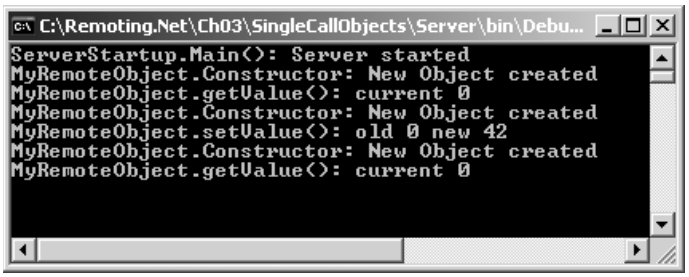

Figure 3-2. *Server's output for a SingleCall object*

This indicates that the server will really create one object for each call.

■ **Note** If you use the .NET Framework version 1.0, you'll see that an additional instance is created for the first remote invocation. This object is used by the framework to check whether the type has been correctly configured and whether it is accessible by the remoting framework. This first instance will immediately be thrown away without doing any work. This behavior is different with version 1.1 of the .NET Framework—in this case, you won't see this additional object creation.

Singleton Objects

Only one instance of a Singleton object can exist at any given time. When receiving a client's request, the server checks its internal tables to see if an instance of this class already exists; if not, this object will be created and stored in the table. After this check the method will be executed. The server guarantees that there will be *exactly one* or no instance available at a given time.

■ **Note** Singletons have an associated lifetime as well, so be sure to override the standard lease time if you don't want your object to be destroyed after some minutes. (More on this later in this chapter.)

For registering an object as a Singleton, you can use the following lines of code:

```
RemotingConfiguration.RegisterWellKnownServiceType(
        typeof(<YourClass>), "<URL>",
        WellKnownObjectMode.Singleton);
```

The ServerStartup class in your sample server will be changed accordingly:

```
class ServerStartup
{
   static void Main(string[] args)
   {
      Console.WriteLine ("ServerStartup.Main(): Server started");
```

```
        HttpChannel chnl = new HttpChannel(1234);
        ChannelServices.RegisterChannel(chnl);
        RemotingConfiguration.RegisterWellKnownServiceType(
              typeof(MyRemoteObject),
              "MyRemoteObject.soap",
              WellKnownObjectMode.Singleton);

        // the server will keep running until keypress.
        Console.ReadLine();
    }
}
```

When the client is started, the output will show a behavior consistent with the "normal" object-oriented way of thinking; the value that is returned is the same value you set two lines before (see Figure 3-3).

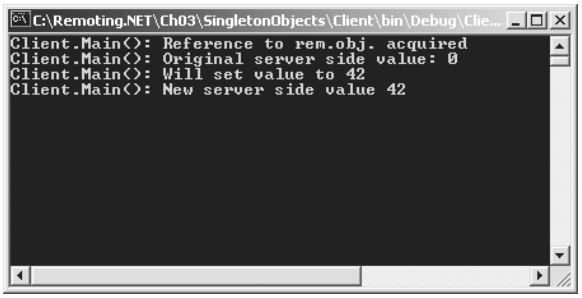
C:\Remoting.NET\Ch03\SingletonObjects\Client\bin\Debug\Clie...
Client.Main(): Reference to rem.obj. acquired
Client.Main(): Original server side value: 0
Client.Main(): Will set value to 42
Client.Main(): New server side value 42

Figure 3-3. *Client's output for a Singleton object*

The same is true for the server, as Figure 3-4 shows.

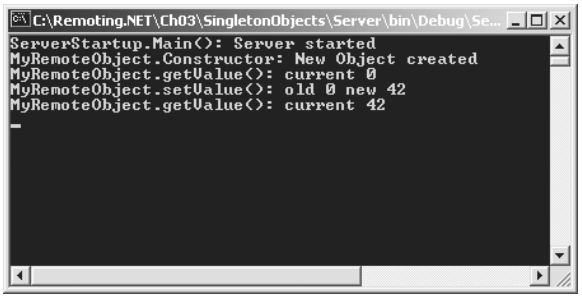

Figure 3-4. *Server's output for a Singleton object*

An interesting thing happens when a second client is started afterwards. This client will receive a value of 42 directly after startup without your setting this value beforehand (see Figures 3-5 and 3-6). This is because only one instance exists at the server, and the instance will stay alive even after the first client is disconnected.

■ **Tip** Use Singletons when you want to share data or resources between clients. But always keep in mind that more than one client might access the same object at any given time, so you have to write the server-side code in a thread-safe way.

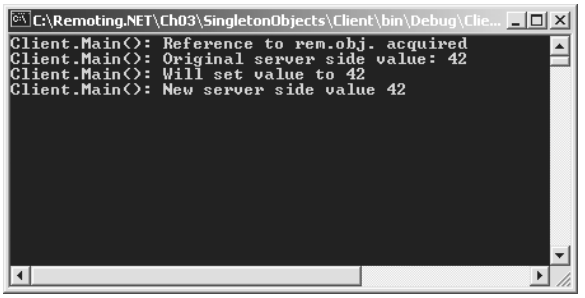
C:\Remoting.NET\Ch03\SingletonObjects\Client\bin\Debug\Clie...
Client.Main(): Reference to rem.obj. acquired
Client.Main(): Original server side value: 42
Client.Main(): Will set value to 42
Client.Main(): New server side value 42

Figure 3-5. *The second client's output when calling a Singleton object*

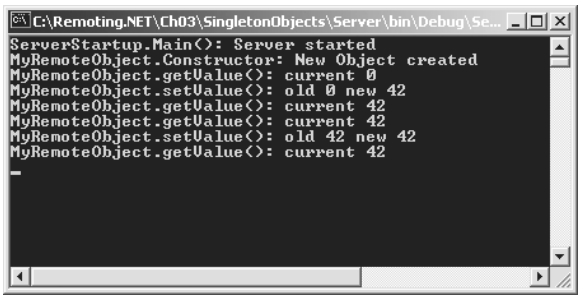

Figure 3-6. *Server's output after the second call to a SingleCall object*

Published Objects

When using either SingleCall or Singleton objects, the necessary instances will be created dynamically during a client's request. When you want to publish a certain object instance that's been precreated on the server—for example, one using a nondefault constructor—neither alternative provides you with a solution.

In this case you can use `RemotingServices.Marshal()` to publish a given instance that behaves like a Singleton afterwards. The only difference is that the object has to already exist at the server before publication.

```
YourObject obj = new YourObject(<your params for constr>);
RemotingServices.Marshal(obj,"YourUrl.soap");
```

The code in the ServerStartup class will look like this:

```
class ServerStartup
{
   static void Main(string[] args)
   {
      Console.WriteLine ("ServerStartup.Main(): Server started");

      HttpChannel chnl = new HttpChannel(1234);
      ChannelServices.RegisterChannel(chnl);

      MyRemoteObject obj = new MyRemoteObject(4711);
      RemotingServices.Marshal(obj,"MyRemoteObject.soap");

      // the server will keep running until keypress.
      Console.ReadLine();
   }
}
```

When the client is run, you can safely expect to get a value of 4711 on the first request because you started the server with this initial value (see Figures 3-7 and 3-8).

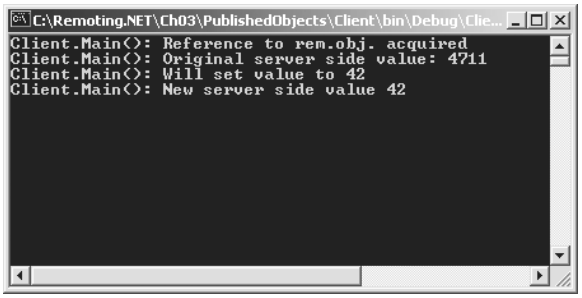
C:\Remoting.NET\Ch03\PublishedObjects\Client\bin\Debug\Clie...
Client.Main(): Reference to rem.obj. acquired
Client.Main(): Original server side value: 4711
Client.Main(): Will set value to 42
Client.Main(): New server side value 42

Figure 3-7. *Client's output when calling a published object*

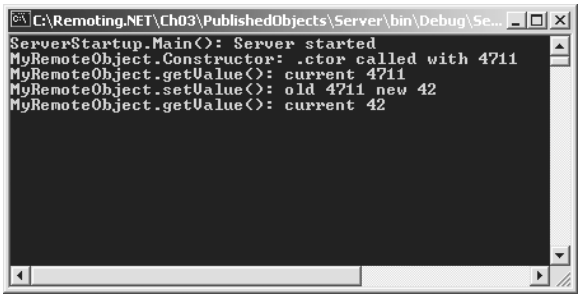

Figure 3-8. *Server's output when publishing the object*

Client-Activated Objects

A client-activated object (CAO) behaves mostly the same way as does a "normal" .NET object (or a COM object).[1] When a creation request on the client is encountered (using `Activator.CreateInstance()` or the `new` operator), an activation message is sent to the server, where an instance of the specified class is created. The server then creates an ObjRef, which is used to uniquely identify this object and returns it to the client. On the client proxy, this ObjRef will be turned into a *TransparentProxy*, which points to the underlying server-side instance.

A client-activated object's lifetime is managed by the same lifetime service used by SAOs, as shown later in this chapter. CAOs are so-called stateful objects; an instance variable that has been set by the client can be retrieved again and will contain the correct value. These objects will store state information from one method call to the other. CAOs are explicitly created by the client, so they can have distinct constructors like normal .NET objects do.

Direct/Transparent Creation

The .NET Remoting framework can be configured to allow client-activated objects to be created like normal objects using the `new` operator. Unfortunately, this manner of creation has one serious drawback: you cannot use shared interfaces or base classes. This means that you either have to ship the compiled objects to your clients or use SoapSuds to extract the metadata.

This tool allows you to extract a metadata-only assembly out of a running server or a server-side implementation assembly. In the past two years, experience has taught me that relying on this tool is not a good choice for most applications. As of today, Microsoft suggests not to use it for .NET to .NET distributed applications. I will nevertheless demonstrate the use of SoapSuds.exe in case you are willing to take the risk.

■ **Caution** If you use the initial version 1.1 of the .NET Framework (without service packs), metadata generated by SoapSuds cannot be used for client-activated objects. This is a bug that has been detailed in article 823445 in the Microsoft Knowledge Base. You can find more details about this problem—and how to contact Product Support Services (PSS) to obtain a hotfix—at `http://support.microsoft.com/default.aspx?scid=kb;en-us;823445`.

In the following example, you'll use more or less the same class you did in the previous examples; it will provide your client with a `SetValue()` and `GetValue()` method to store and retrieve an int value as the object's state. The metadata that is needed for the client to create a reference to the CAO will be extracted with SoapSuds.exe, about which you'll read more later in this chapter.

The reliance on SoapSuds allows you to develop the server application without any need for up-front design of a shared assembly, therefore the server will simply include the CAOs implementation. You can see this in Listing 3-4.

---

1. The only exception from this rule lies in the object's lifetime, which is managed completely differently from the way it is in .NET generally or in COM.

Listing 3-4. *A Server That Offers a Client-Activated Object*

```
using System;
using System.Runtime.Remoting;
using System.Runtime.Remoting.Channels.Http;
using System.Runtime.Remoting.Channels;

namespace Server
{

   public class MyRemoteObject: MarshalByRefObject
   {
      int myvalue;

      public MyRemoteObject(int val)
      {
         Console.WriteLine("MyRemoteObject.ctor(int) called");
         myvalue = val;
      }

      public MyRemoteObject()
      {
         Console.WriteLine("MyRemoteObject.ctor() called");
      }

      public void SetValue(int newval)
      {
         Console.WriteLine("MyRemoteObject.SetValue(): old {0} new {1}",
                           myvalue,newval);
         myvalue = newval;
      }

      public int GetValue()
      {
         Console.WriteLine("MyRemoteObject.GetValue(): current {0}",myvalue);
         return myvalue;
      }
   }

   class ServerStartup
   {
      static void Main(string[] args)
      {
         Console.WriteLine ("ServerStartup.Main(): Server started");

         HttpChannel chnl = new HttpChannel(1234);
         ChannelServices.RegisterChannel(chnl);
```

```
            RemotingConfiguration.ApplicationName = "MyServer";
            RemotingConfiguration.RegisterActivatedServiceType(
                              typeof(MyRemoteObject));

            // the server will keep running until keypress.
            Console.ReadLine();
        }
    }
}
```

On the server you now have the new startup code needed to register a channel and this class as a client-activated object. When adding a Type to the list of activated services, you cannot provide a single URL for each object; instead, you have to set `RemotingConfiguration.ApplicationName` to a string value that identifies your server.

The URL to your remote object will be `http://<hostname>:<port>/<ApplicationName>`. What happens behind the scenes is that a general activation SAO is automatically created by the framework and published at the URL `http://<hostname>:<port>/<ApplicationName>/RemoteActivationService.rem`. This SAO will take the clients' requests to create a new instance and pass it on to the remoting framework.

To extract the necessary interface information, you can run the following SoapSuds command line in the directory where the server.exe assembly has been placed:

```
soapsuds -ia:server -nowp -oa:generated_metadata.dll
```

■ **Note** You should perform all command-line operations from the Visual Studio command prompt, which you can bring up by selecting Start ➤ All Programs ➤ Microsoft Visual Studio .NET 2003 ➤ Visual Studio .NET Tools. This command prompt sets the correct "path" variable to include the .NET SDK tools.

The resulting generated_metadata.dll assembly must be referenced by the client. The sample client also registers the CAO and acquires two references to (different) remote objects. It then sets the value of those objects and outputs them again, which shows that you really are dealing with two different objects.

As you can see in Listing 3-5, the activation of the remote object is done with the `new` operator. This is possible because you registered the Type as ActivatedClientType before. The runtime now knows that whenever your application creates an instance of this class, it instead should create a reference to a remote object running on the server.

Listing 3-5. *The Client Accesses the Client-Activated Object*

```
using System;
using System.Runtime.Remoting;
using System.Runtime.Remoting.Channels.Http;
using System.Runtime.Remoting.Channels;
using System.Runtime.Remoting.Activation;
using Server;
```

```
namespace Client
{
   class Client
   {
      static void Main(string[] args)
      {
         HttpChannel channel = new HttpChannel();
         ChannelServices.RegisterChannel(channel);

         RemotingConfiguration.RegisterActivatedClientType(
            typeof(MyRemoteObject),
            "http://localhost:1234/MyServer");

         Console.WriteLine("Client.Main(): Creating first object");
         MyRemoteObject obj1 = new MyRemoteObject();
         obj1.SetValue(42);

         Console.WriteLine("Client.Main(): Creating second object");
         MyRemoteObject obj2 = new MyRemoteObject();
         obj2.SetValue(4711);

         Console.WriteLine("Obj1.GetValue(): {0}",obj1.GetValue());
         Console.WriteLine("Obj2.GetValue(): {0}",obj2.GetValue());

         Console.ReadLine();
      }
   }
}
```

When this code sample is run, you will see the same behavior as when using local objects—the two instances have their own state (Figure 3-9). As expected, on the server two different objects are created (Figure 3-10).

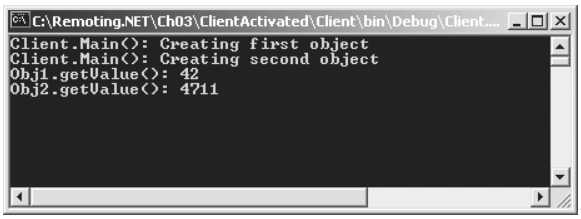

Figure 3-9. *Client-side output when using CAOs*

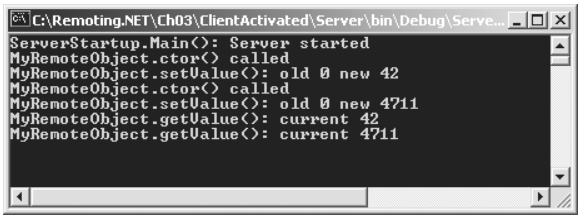

Figure 3-10. *Server-side output when using CAOs*

Using the Factory Design Pattern

From what you've read up to this point, you know that the reliance on SoapSuds might not be the best choice for your distributed application. You can instead use a ffactory design pattern, in which you'll include a SAO providing methods that return new instances of the CAO.

■ **Note** You might also just ship the server-side implementation assembly to the client and reference it directly. But as I stated previously, this is clearly against all distributed application design principles and will lead to a number of versioning and deployment issues.

Here, I just give you a short introduction to the factory design pattern. Basically you have two classes, one of which is a factory, and the other is the real object you want to use. Due to constraints of the real class, you will not be able to construct it directly, but instead will have to call a method on the factory, which creates a new instance and passes it to the client.

Listing 3-6 shows you a fairly simple implementation of this design pattern.

Listing 3-6. *The Factory Design Pattern*

```
using System;

namespace FactoryDesignPattern
{

   class MyClass
   {
   }

   class MyFactory
   {
      public MyClass GetNewInstance()
      {
         return new MyClass();
      }
   }
```

```
   class MyClient
   {
      static void Main(string[] args)
      {
         // creation using "new"
         MyClass obj1 = new MyClass();

         // creating using a factory
         MyFactory fac = new MyFactory();
         MyClass obj2 = fac.GetNewInstance();

      }
   }

}
```

When bringing this pattern to remoting, you have to create a factory that's running as a server-activated object (ideally a Singleton) that has a method returning a new instance of the "real class" (the CAO) to the client. This gives you a huge advantage in that you don't have to distribute the implementation to the client system and still can avoid using SoapSuds.

■ **Note** Distributing the implementation to the client is not only a bad choice due to deployment issues, it also makes it possible for the client user to disassemble your object's codes using ILDASM or some other tool.

You have to design your factory SAO using a shared assembly that contains the interface information (or abstract base classes) which are implemented by your remote objects. This is shown in Listing 3-7.

Listing 3-7. *The Shared Interfaces for the Factory Design Pattern*

```
using System;

namespace General
{
   public interface IRemoteObject
   {
      void SetValue(int newval);
      int GetValue();
   }

   public interface IRemoteFactory
   {
      IRemoteObject GetNewInstance();
      IRemoteObject GetNewInstance(int initvalue);
   }
}
```

On the server you now have to implement both interfaces and create a startup code that registers the factory as a SAO. You don't have to register the CAO in this case because every MarshalByRefObject can be returned by a method call; the framework takes care of the necessity to remote each call itself, as shown in Listing 3-8.

Listing 3-8. *The Server-Side Factory Pattern's Implementation*

```
using System;
using System.Runtime.Remoting;
using System.Runtime.Remoting.Channels.Http;
using System.Runtime.Remoting.Channels;
using System.Runtime.Remoting.Messaging;
using General;

namespace Server
{
   class MyRemoteObject: MarshalByRefObject, IRemoteObject
   {
      int myvalue;

      public MyRemoteObject(int val)
      {
         Console.WriteLine("MyRemoteObject.ctor(int) called");
         myvalue = val;
      }

      public MyRemoteObject()
      {
         Console.WriteLine("MyRemoteObject.ctor() called");
      }

      public void SetValue(int newval)
      {
         Console.WriteLine("MyRemoteObject.SetValue(): old {0} new {1}",
                           myvalue,newval);
         myvalue = newval;
      }

      public int GetValue()
      {
         Console.WriteLine("MyRemoteObject.GetValue(): current {0}",myvalue);
         return myvalue;
      }
   }
```

```
   class MyRemoteFactory: MarshalByRefObject,IRemoteFactory
   {
      public MyRemoteFactory() {
         Console.WriteLine("MyRemoteFactory.ctor() called");
      }

      public IRemoteObject GetNewInstance()
      {
         Console.WriteLine("MyRemoteFactory.GetNewInstance() called");
         return new MyRemoteObject();
      }

      public IRemoteObject getNewInstance(int initvalue)
      {
         Console.WriteLine("MyRemoteFactory.getNewInstance(int) called");
         return new MyRemoteObject(initvalue);
      }
   }


   class ServerStartup
   {
      static void Main(string[] args)
      {
         Console.WriteLine ("ServerStartup.Main(): Server started");

         HttpChannel chnl = new HttpChannel(1234);
         ChannelServices.RegisterChannel(chnl);

         RemotingConfiguration.RegisterWellKnownServiceType(
            typeof(MyRemoteFactory),
            "factory.soap",
            WellKnownObjectMode.Singleton);

         // the server will keep running until keypress.
         Console.ReadLine();
      }
   }
}
```

The client, which is shown in Listing 3-9, works a little bit differently from the previous one as well. It creates a reference to a remote SAO using `Activator.GetObject()`, upon which it places two calls to `GetNewInstance()` to acquire two different remote CAOs.

Listing 3-9. *The Client Uses the Factory Pattern*

```
using System;
using System.Runtime.Remoting;
using System.Runtime.Remoting.Channels.Http;
```

```
using System.Runtime.Remoting.Channels.Tcp;
using System.Runtime.Remoting.Channels;
using General;

namespace Client
{
   class Client
   {
      static void Main(string[] args)
      {
         HttpChannel channel = new HttpChannel();
         ChannelServices.RegisterChannel(channel);

         Console.WriteLine("Client.Main(): Creating factory");
         IRemoteFactory fact = (IRemoteFactory) Activator.GetObject(
            typeof(IRemoteFactory),
            "http://localhost:1234/factory.soap");

         Console.WriteLine("Client.Main(): Acquiring first object from factory");
         IRemoteObject obj1 = fact.GetNewInstance();
         obj1.SetValue(42);

         Console.WriteLine("Client.Main(): Acquiring second object from " +
                           "factory");
         IRemoteObject obj2 = fact.GetNewInstance(4711);

         Console.WriteLine("Obj1.GetValue(): {0}",obj1.GetValue());
         Console.WriteLine("Obj2.GetValue(): {0}",obj2.GetValue());

         Console.ReadLine();
      }
   }
}
```

When this sample is running, you see that the client behaves nearly identically to the previous example, but the second object's value has been set using the object's constructor, which is called via the factory (Figure 3-11). On the server a factory object is generated, and each new instance is created using the overloaded method `GetNewInstance()` (Figure 3-12).

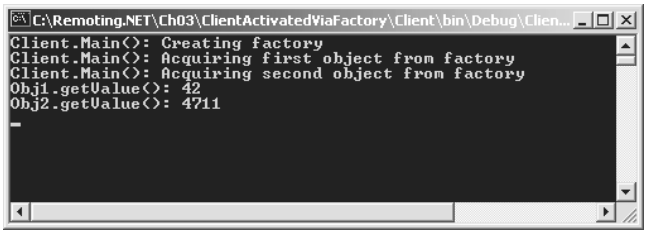

Figure 3-11. *Client-side output when using a factory object*

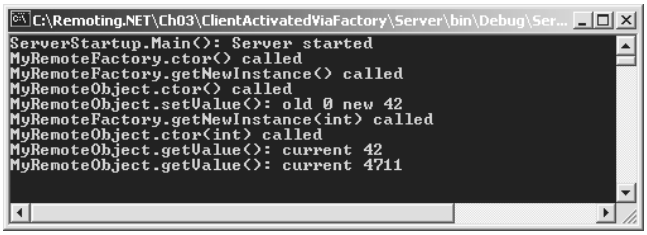

Figure 3-12. *Server-side output when using a factory object*

Managing Lifetime

One point that can lead to a bit of confusion is the way an object's lifetime is managed in the .NET Remoting framework. Common .NET objects are managed using a garbage collection algorithm that checks if any other object is still using a given instance. If not, the instance will be garbage collected and disposed.

If you would apply a similar scheme to remote objects, it would mean to ping the client-side proxies to ensure that they are still using the objects and that the application is still running. This is mainly what DCOM did. The reason for this is that normally a client that has been closed unexpectedly or went offline due to a network outage might not have decremented the server-side reference counter. Without some additional measure, these server-side objects would in turn use the server's resources forever. Unfortunately, when your client is behind an HTTP proxy and is accessing your objects using SOAP remoting, the server will not be able to contact the client in any way.

This constraint leads to a new kind of lifetime service: the lease-based object lifetime. Basically this means that each server-side object is associated with a lease upon creation. This lease will have a time-to-live counter (which starts at five minutes by default) that is decremented in certain intervals. In addition to the initial time, a defined amount (two minutes in the default configuration) is added to this time to live upon every method call a client places on the remote object.

When this time reaches zero, the framework looks for any sponsors registered with this lease. A sponsor is an object running on the server itself, the client, or any machine reachable via a network that will take a call from the .NET Remoting framework asking whether an object's lifetime should be renewed or not (more on this in Chapter 6).

When the sponsor decides that the lease will not be renewed or when the framework is unable to contact any of the registered sponsors, the object is marked as timed out and will be subject to garbage collection. When a client still has a reference to a timed-out object and calls a method on it, it will receive an exception.

To change the default lease times, you can override `InitializeLifetimeService()` in the MarshalByRefObject. In the following example, you see how to change the previous CAO sample to implement a different lifetime of only ten milliseconds for this object. Normally LeaseManager only polls all leases every ten seconds, so you have to change this polling interval as well.

```
using System.Runtime.Remoting.Lifetime
namespace Server
{
   class MyRemoteObject: MarshalByRefObject, IRemoteObject
   {
```

```
      public override object InitializeLifetimeService()
      {
         Console.WriteLine("MyRemoteObject.InitializeLifetimeService() called");
         ILease lease = (ILease)base.InitializeLifetimeService();
         if (lease.CurrentState == LeaseState.Initial)
         {
            lease.InitialLeaseTime = TimeSpan.FromMilliseconds(10);
            lease.SponsorshipTimeout = TimeSpan.FromMilliseconds(10);
            lease.RenewOnCallTime = TimeSpan.FromMilliseconds(10);
         }
         return lease;
      }

      // rest of implementation ...
   }

   class MyRemoteFactory: MarshalByRefObject,IRemoteFactory
   {
       // rest of implementation ...
   }

   class ServerStartup
   {
      static void Main(string[] args)
      {
         Console.WriteLine ("ServerStartup.Main(): Server started");

         LifetimeServices.LeaseManagerPollTime = TimeSpan.FromMilliseconds(10);

         HttpChannel chnl = new HttpChannel(1234);
         ChannelServices.RegisterChannel(chnl);

         RemotingConfiguration.RegisterWellKnownServiceType(
            typeof(MyRemoteFactory),
            "factory.soap",
            WellKnownObjectMode.Singleton);

         // the server will keep running until keypress.
         Console.ReadLine();
      }
   }
}
```

On the client side, you can add a one-second delay between creation and the first call on the remote object to see the effects of the changed lifetime. You also need to provide some code to handle the RemotingException that will get thrown because the object is no longer available at the server. The client is shown in Listing 3-10.

Listing 3-10. *A Client That Calls a Timed-Out CAO*

```
using System;
using System.Runtime.Remoting;
using System.Runtime.Remoting.Channels.Http;
using System.Runtime.Remoting.Channels.Tcp;
using System.Runtime.Remoting.Channels;
using General;

namespace Client
{
   class Client
   {
      static void Main(string[] args)
      {
         HttpChannel channel = new HttpChannel();
         ChannelServices.RegisterChannel(channel);

         Console.WriteLine("Client.Main(): Creating factory");
         IRemoteFactory fact = (IRemoteFactory) Activator.GetObject(
            typeof(IRemoteFactory),
            "http://localhost:1234/factory.soap");

         Console.WriteLine("Client.Main(): Acquiring object from factory");
         IRemoteObject obj1 = fact.GetNewInstance();

         Console.WriteLine("Client.Main(): Sleeping one second");
         System.Threading.Thread.Sleep(1000);

         Console.WriteLine("Client.Main(): Setting value");
         try
         {
            obj1.SetValue(42);
         }
         catch (Exception e)
         {
            Console.WriteLine("Client.Main(). EXCEPTION \n{0}",e.Message);
         }

         Console.ReadLine();
      }
   }
}
```

Running this sample, you see that the client is able to successfully create a factory object and call its `GetNewInstance()` method (Figure 3-13). When calling `SetValue()` on the returned CAO, the client will receive an exception stating the object has timed out. The server runs normally (Figure 3-14).

■ **Note** If you use version 1.0 of the .NET Framework, this exception's text will look different as it includes the GUID that has been used to identify the remote service.

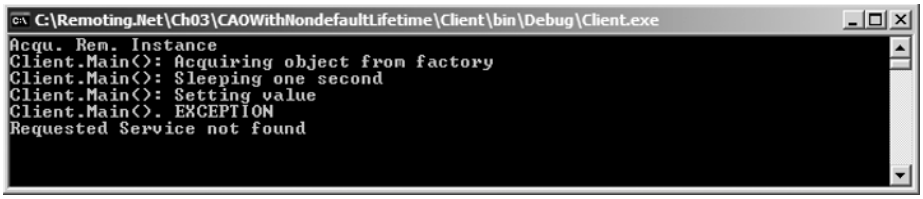

Figure 3-13. *The client receives an exception because the object has timed out.*

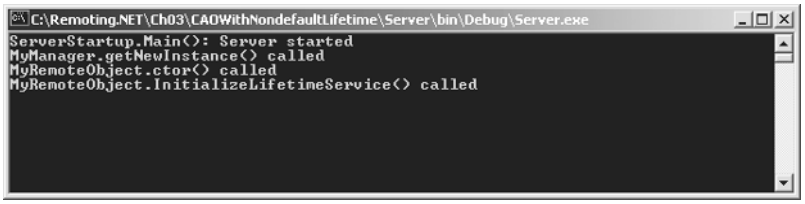

Figure 3-14. *The server when overriding InitializeLifetimeService()*

Types of Invocation

The .NET Framework provides three possibilities to call methods on remote objects (no matter if they are Singleton, SingleCall, or published objects). You can execute their methods in a synchronous, asynchronous, or one-way fashion.

Synchronous calls are basically what I showed you in the preceding examples. The server's remote method is called like a common method, and the client blocks and waits until the server has completed its processing. If an exception occurs during execution of the remote invocation, the exception is thrown at the line of code in which you called the server.

Asynchronous calls are executed in a two-step process.[2] The first step triggers the execution but does not wait for the method's response value. The program flow continues on the client. When you are ready to collect the server-side function's response, you have to call another method that checks whether the server has already finished processing your request; if not, it blocks until finalization. Any exception thrown during the call of your method will be rethrown at the line of code where you collect the response. Even if the server has been offline, you won't be notified beforehand.

TThe last kind of function is a little different from the preceding ones. With one-way methods, you don't have the option of receiving return values or getting an exception if the server has been offline or otherwise unable to fulfill your request. The .NET Remoting framework will just *try* to call the methods on the remote server and won't do anything else.

2. For the in-depth story about asynchronous calls, please refer to Chapter 7.

Synchronous Calls

As I've mentioned, synchronous calls are the usual way of calling a function in the .NET Framework. The server will be contacted directly and, except when using multiple client-side threads, the client code will block until the server has finished executing its method. If the server is unavailable or an exception occurs while carrying out your request, the exception will be rethrown at the line of code where you called the remote method.

Using Synchronous Calls

In the following series of examples for the different types of invocation, you use a common server and a shared assembly called General.dll (you'll see some slight modifications in the last part). This server just provides you with a Singleton object that stores an int as its state and has an additional method that returns a String. You'll use this later to demonstrate the collection of return values when using asynchronous calls.

Defining the General.dll

In Listing 3-11, you see the shared General.dll in which the necessary interface is defined.

Listing 3-11. *The Shared Assembly's Source Code*

```
using System;
using System.Runtime.Remoting.Messaging;

namespace General
{
   public interface IMyRemoteObject
   {
      void SetValue(int newval);
      int GetValue();
      String GetName();
   }
}
```

Creating the Server

The server, shown in Listing 3-12, implements the defined methods with the addition of making the `SetValue()` and `GetName()` functions long-running code. In both methods, a five-second delay is introduced so you can see the effects of long-lasting execution in the different invocation contexts.

Listing 3-12. *A Server with Some Long-Running Methods*

```
using System;
using System.Runtime.Remoting;
using General;
using System.Runtime.Remoting.Channels.Http;
using System.Runtime.Remoting.Channels;
using System.Runtime.Remoting.Messaging;
```

```
using System.Collections;
using System.Threading;

namespace Server
{
   class MyRemoteObject: MarshalByRefObject, IMyRemoteObject
   {
      int myvalue;

      public MyRemoteObject()
      {
         Console.WriteLine("MyRemoteObject.Constructor: New Object created");
      }

      public void SetValue(int newval)
      {
         Console.WriteLine("MyRemoteObject.setValue(): old {0} new {1}",
                           myvalue,newval);

         // we simulate a long running action
         Console.WriteLine("     .setValue() -> waiting 5 sec before setting" +
                           " value");
         Thread.Sleep(5000);

         myvalue = newval;
         Console.WriteLine("     .SetValue() -> value is now set");
      }

      public int GetValue()
      {
         Console.WriteLine("MyRemoteObject.GetValue(): current {0}",myvalue);
         return myvalue;
      }

      public String GetName()
      {
         Console.WriteLine("MyRemoteObject.getName(): called");

         // we simulate a long running action
         Console.WriteLine("     .GetName() -> waiting 5 sec before continuing");
         Thread.Sleep(5000);

         Console.WriteLine("     .GetName() -> returning name");
         return "John Doe";
      }
   }
```

```
   class ServerStartup
   {
      static void Main(string[] args)
      {
         Console.WriteLine ("ServerStartup.Main(): Server started");

         HttpChannel chnl = new HttpChannel(1234);
         ChannelServices.RegisterChannel(chnl);

         RemotingConfiguration.RegisterWellKnownServiceType(
               typeof(MyRemoteObject),
               "MyRemoteObject.soap",
               WellKnownObjectMode.Singleton);

         // the server will keep running until keypress.
         Console.ReadLine();
      }

   }
}
```

Creating the Client

The first client, which is shown in Listing 3-13, calls the server synchronously, as in all preceding examples. It calls all three methods and gives you statistics on how long the total execution took.

Listing 3-13. *The First Client Calls the Methods Synchronously*

```
using System;
using System.Runtime.Remoting;
using General;
using System.Runtime.Remoting.Channels.Http;
using System.Runtime.Remoting.Channels.Tcp;
using System.Runtime.Remoting.Channels;
using System.Runtime.Remoting.Proxies;
using System.Threading;
using System.Reflection;

namespace Client
{

   class Client
   {

      static void Main(string[] args)
      {
         DateTime start = System.DateTime.Now;
```

```
            HttpChannel channel = new HttpChannel();
            ChannelServices.RegisterChannel(channel);
            IMyRemoteObject obj = (IMyRemoteObject) Activator.GetObject(
               typeof(IMyRemoteObject),
               "http://localhost:1234/MyRemoteObject.soap");
            Console.WriteLine("Client.Main(): Reference to rem.obj. acquired");

            Console.WriteLine("Client.Main(): Will set value to 42");

            obj.SetValue(42);

            Console.WriteLine("Client.Main(): Will now read value");
            int tmp = obj.GetValue();
            Console.WriteLine("Client.Main(): New server side value {0}", tmp);


            Console.WriteLine("Client.Main(): Will call getName()");
            String name = obj.GetName();
            Console.WriteLine("Client.Main(): received name {0}",name);

            DateTime end = System.DateTime.Now;
            TimeSpan duration = end.Subtract(start);
            Console.WriteLine("Client.Main(): Execution took {0} seconds.",
                           duration.Seconds);

            Console.ReadLine();
        }
    }
}
```

As the calls to the long-running methods `GetName()` and `SetValue()` are expected to take roughly five seconds each, and you have to add a little overhead for .NET Remoting (especially for the first call on a remote object), this example will take more than ten seconds to run.

You can see that this assumption is right by looking at the client's output in Figure 3-15. The total client execution takes 12 seconds. When looking at the server's output in Figure 3-16, note that all methods are called synchronously. Every method is finished before the next one is called by the client.

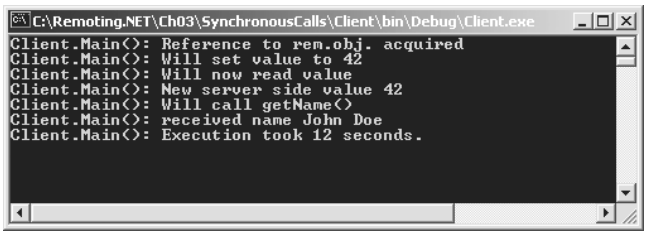
C:\Remoting.NET\Ch03\SynchronousCalls\Client\bin\Debug\Client.exe
Client.Main(): Reference to rem.obj. acquired
Client.Main(): Will set value to 42
Client.Main(): Will now read value
Client.Main(): New server side value 42
Client.Main(): Will call getName()
Client.Main(): received name John Doe
Client.Main(): Execution took 12 seconds.

Figure 3-15. *Client's output when using synchronous calls*

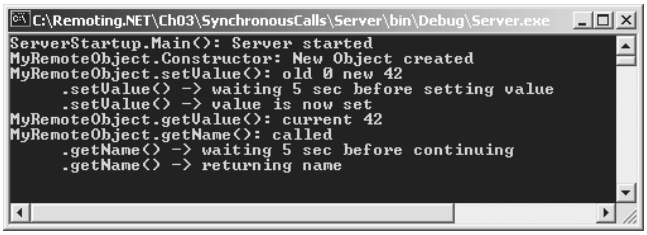
C:\Remoting.NET\Ch03\SynchronousCalls\Server\bin\Debug\Server.exe
ServerStartup.Main(): Server started
MyRemoteObject.Constructor: New Object created
MyRemoteObject.setValue(): old 0 new 42
.setValue() -> waiting 5 sec before setting value
.setValue() -> value is now set
MyRemoteObject.getValue(): current 42
MyRemoteObject.getName(): called
.getName() -> waiting 5 sec before continuing
.getName() -> returning name

Figure 3-16. *Server's output when called synchronously*

Asynchronous Calls

In the synchronous calls example, you saw that waiting for every method to complete incurs a performance penalty if the calls themselves are independent; the second call doesn't need the output from the first. You could now use a separate thread to call the second method, but even though threading is quite simple in .NET, it would probably render the application more complex if you use a distinct thread for any longer lasting remote function call. The .NET Framework provides a feature, called asynchronous delegates, that allows methods to be called in an asynchronous fashion with only three lines of additional code.

Delegate Basics

A delegate is, in its regular sense, just a kind of an object-oriented function pointer. You will initialize it and pass a function to be called when the delegate is invoked. In .NET Framework, a delegate is a subclass of `System.MulticastDelegate`, but C# provides an easier way to define a delegate instead of declaring a new `Class`.

Declaring a Delegate

The declaration of a delegate looks quite similar to the declaration of a method.

```
delegate <ReturnType> <name> ([parameters]);
```

As the delegate will call a method at some point in the future, you have to provide it with a declaration that matches the method's signature. When you want a delegate to call the following method:

```
public String DoSomething(int myValue)
```

you have to define it as follows:

```
delegate String DoSomethingDelegate (int myValue);
```

■ **Note** The delegate's parameter and return types have to match those of the method.

Remember that the delegate is in reality just another class, so you cannot define it within a method's body, only directly within a namespace or another class!

Asynchronously Invoking a Delegate

When you want to use a delegate, you first have to create an instance of it, passing the method to be called as a constructor parameter:

```
DoSomethingDelegate del = new DoSomethingDelegate(DoSomething);
```

■ **Note** When passing the method to the constructor, be sure not to include an opening or closing parenthesis— (or) —as in `DoSomething()`. The previous example uses a static method `DoSomething` in the same class. When using static methods of other classes, you have to pass `SomeClass.SomeMethod`, and for instance methods, you pass `SomeObject.DoSomething`.

The asynchronous invocation of a delegate is a two-step process. In the first step, you have to trigger the execution using `BeginInvoke()`, as follows:

```
IAsyncResult ar = del.BeginInvoke(42,null,null);
```

■ **Note** If you use Microsoft Visual Studio 2002, `BeginInvoke()` behaves a little strangely in the IDE. You won't see it using IntelliSense, as it is automatically generated during compilation. The parameters are the same as the method parameters, according to the delegate definition, followed by two other objects; you won't be using these two objects in the following examples, instead passing null to `BeginInvoke()`. In Visual Studio 2003, you will see the complete information in the IDE.

`BeginInvoke()` then returns an IAsyncResult object that will be used later to retrieve the method's return values. When ready to do so, you call `EndInvoke()` on the delegate passing the IAsyncResult as a parameter. The `EndInvoke()` method will block until the server has completed executing the underlying method.

```
String res = del.EndInvoke(ar);
```

■ **Note** `EndInvoke()` will not be visible in the Visual Studio 2002 IDE either. The method takes an IAsyncResult as a parameter, and its return type will be defined in the delegate's declaration.

Creating an Example Delegate

In Listing 3-14, a delegate is used to asynchronously call a local function and wait for its result. The method returns a String built from the passed int parameter.

Listing 3-14. *Using a Delegate in a Local Application*

```
using System;

namespace SampleDelegate
{
```

```
   class SomethingClass
   {
      delegate String DoSomethingDelegate(int myValue);

      public static String DoSomething(int myValue)
      {
         return "HEY:" + myValue.ToString();
      }

      static void Main(string[] args)
      {
         DoSomethingDelegate del = new DoSomethingDelegate(DoSomething);
         IAsyncResult ar = del.BeginInvoke(42,null,null);
         // ... do something different here
         String res = del.EndInvoke(ar);

         Console.WriteLine("Got result: '{0}'",res);

         // wait for return to close
         Console.ReadLine();
      }
   }
}
```

As expected, the application outputs "HEY:42" as you can see in Figure 3-17.

Figure 3-17. *The sample delegate*

Implementing the New Client

In the new remoting client, shown in Listing 3-15, you see how to change the calls to `GetName()` and `SetValue()` to use delegates as well. Your client then invokes both delegates and subsequently waits for their completion before synchronously calling `GetValue()` on the server. In this instance, you use the same server application as in the preceding example.

Listing 3-15. *The New Client Now Using Asynchronous Delegates*

```
using System;
using System.Runtime.Remoting;
using General;
using System.Runtime.Remoting.Channels.Http;
using System.Runtime.Remoting.Channels.Tcp;
```

```
using System.Runtime.Remoting.Channels;
using System.Runtime.Remoting.Proxies;
using System.Threading;

namespace Client
{

   class Client
   {
      delegate void SetValueDelegate(int value);
      delegate String GetNameDelegate();

      static void Main(string[] args)
      {
         DateTime start = System.DateTime.Now;

         HttpChannel channel = new HttpChannel();
         ChannelServices.RegisterChannel(channel);
         IMyRemoteObject obj = (IMyRemoteObject) Activator.GetObject(
            typeof(IMyRemoteObject),
            "http://localhost:1234/MyRemoteObject.soap");
         Console.WriteLine("Client.Main(): Reference to rem.obj. acquired");

         Console.WriteLine("Client.Main(): Will call setValue(42)");
         SetValueDelegate svDelegate = new SetValueDelegate(obj.SetValue);
         IAsyncResult svAsyncres = svDelegate.BeginInvoke(42,null,null);
         Console.WriteLine("Client.Main(): Invocation done");

         Console.WriteLine("Client.Main(): Will call GetName()");
         GetNameDelegate gnDelegate = new GetNameDelegate(obj.GetName);
         IAsyncResult gnAsyncres = gnDelegate.BeginInvoke(null,null);
         Console.WriteLine("Client.Main(): Invocation done");

         Console.WriteLine("Client.Main(): EndInvoke for SetValue()");
         svDelegate.EndInvoke(svAsyncres);
         Console.WriteLine("Client.Main(): EndInvoke for SetName()");
         String name = gnDelegate.EndInvoke(gnAsyncres);

         Console.WriteLine("Client.Main(): received name {0}",name);

         Console.WriteLine("Client.Main(): Will now read value");
         int tmp = obj.GetValue();
         Console.WriteLine("Client.Main(): New server side value {0}", tmp);

         DateTime end = System.DateTime.Now;
         TimeSpan duration = end.Subtract(start);
```

```
            Console.WriteLine("Client.Main(): Execution took {0} seconds.",
                              duration.Seconds);

            Console.ReadLine();
        }
    }
}
```

When looking in the client's output in Figure 3-18, you can see that both long-running methods have been called at nearly the same time. This results in improved runtime performance, taking the execution time down from 12 seconds to 8 at the expense of making the application slightly more complex.

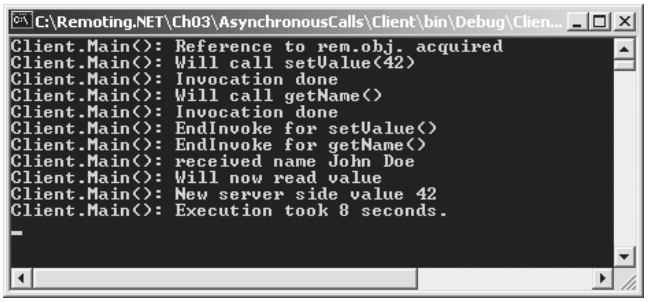
C:\Remoting.NET\Ch03\AsynchronousCalls\Client\bin\Debug\Clien...
Client.Main(): Reference to rem.obj. acquired
Client.Main(): Will call setValue(42)
Client.Main(): Invocation done
Client.Main(): Will call getName()
Client.Main(): Invocation done
Client.Main(): EndInvoke for setValue()
Client.Main(): EndInvoke for getName()
Client.Main(): received name John Doe
Client.Main(): Will now read value
Client.Main(): New server side value 42
Client.Main(): Execution took 8 seconds.

Figure 3-18. *Client output when using asynchronous calls*

The server output in Figure 3-19 shows that both methods have been entered on the server at the same time without blocking the client.

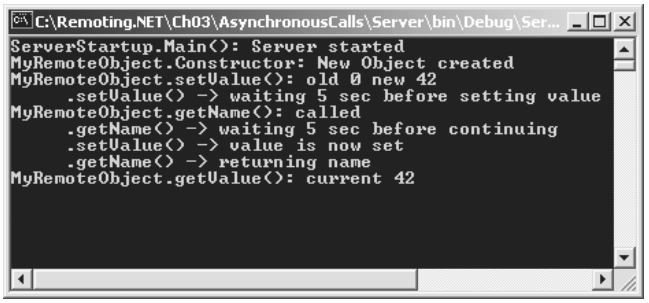
C:\Remoting.NET\Ch03\AsynchronousCalls\Server\bin\Debug\Ser...
ServerStartup.Main(): Server started
MyRemoteObject.Constructor: New Object created
MyRemoteObject.setValue(): old 0 new 42
.setValue() -> waiting 5 sec before setting value
MyRemoteObject.getName(): called
.getName() -> waiting 5 sec before continuing
.setValue() -> value is now set
.getName() -> returning name
MyRemoteObject.getValue(): current 42

Figure 3-19. *Server's output when called asynchronously*

One-Way Calls

One-way calls are a little different from asynchronous calls in the respect that the .NET Framework does not guarantee their execution. In addition, the methods used in this kind of call cannot have return values or out parameters. You can also use delegates to call one-way methods asynchronously, but the `EndInvoke()` function will exit immediately without checking whether the server has finished processing yet. No exceptions are thrown, even if the remote server is

down or the method call is malformed. Reasons for using these kind of methods (which aren't guaranteed to be executed at all) can be found in uncritical logging or tracing facilities, where the nonexistence of the server should not slow down the application.

Demonstrating a One-Way Call

You define one-way methods using the [OneWay] attribute. This happens in the defining metadata (in the General.dll in these examples) and doesn't need a change in the server or the client.

Defining the General.dll

The attribute [OneWay()] has to be specified in the interface definition of each method that will be called this way. As shown in Listing 3-16, you change only the `SetValue()` method to become a one-way method; the others are still defined as earlier.

Listing 3-16. *The Shared Interfaces DLL Defines the One-Way Method.*

```
using System;
using System.Runtime.Remoting.Messaging;

namespace General
{
   public interface IMyRemoteObject
   {
      [OneWay()]
      void SetValue(int newval);
      int GetValue();
      String GetName();
   }
}
```

Implementing the Client

On the server side, no change is needed, so you can directly look at the client. In theory, no modification is needed for the client as well, but extend it a little here to catch the eventual exception during execution, as shown in Listing 3-17.

Listing 3-17. *Try/Catch Blocks Are Added to the Client.*

```
using System;
using System.Runtime.Remoting;
using General;
using System.Runtime.Remoting.Channels.Http;
using System.Runtime.Remoting.Channels.Tcp;
using System.Runtime.Remoting.Channels;
using System.Runtime.Remoting.Proxies;
using System.Threading;

namespace Client
{
```

```
   class Client
   {
      static void Main(string[] args)
      {
         HttpChannel channel = new HttpChannel();
         ChannelServices.RegisterChannel(channel);
         IMyRemoteObject obj = (IMyRemoteObject) Activator.GetObject(
            typeof(IMyRemoteObject),
            "http://localhost:1234/MyRemoteObject.soap");
         Console.WriteLine("Client.Main(): Reference to rem.obj. acquired");

         Console.WriteLine("Client.Main(): Will call SetValue(42)");
         try
         {
            obj.SetValue(42);
            Console.WriteLine("Client.Main(): Value successfully set.");
         }
         catch (Exception e)
         {
            Console.WriteLine("Client.Main(): EXCEPTION.\n{0}",e.Message);
         }
         // wait for keypress
         Console.ReadLine();
      }
   }
}
```

When this client is started, you will see the output in Figure 3-20 *no matter whether the server is running or not.*

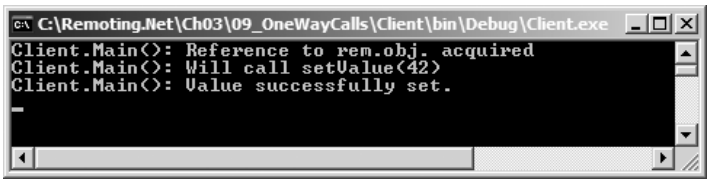

Figure 3-20. *Client output when using one-way methods*

As shown in Listing 3-18, you can now change the method in General.dll back to a standard method (non-one-way) by commenting out the [OneWay()] attribute.

Listing 3-18. *Removing the [OneWay()] Attribute*

```
using System;
using System.Runtime.Remoting.Messaging;
```

```
namespace General
{
   public interface IMyRemoteObject
   {
      // no more oneway attribute [OneWay()]
      void SetValue(int newval);
      int GetValue();
      String GetName();
   }
}
```

Recompilation and a restart of the client (still without a running server) yields the result in Figure 3-21: an exception is thrown and a corresponding error message is output.

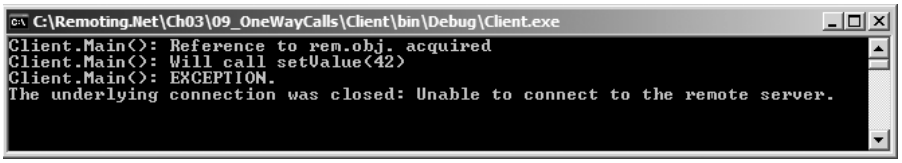
C:\Remoting.Net\Ch03\09_OneWayCalls\Client\bin\Debug\Client.exe
Client.Main(): Reference to rem.obj. acquired
Client.Main(): Will call setValue(42)
Client.Main(): EXCEPTION.
The underlying connection was closed: Unable to connect to the remote server.

Figure 3-21. *Client output when removing the [OneWay()] attribute*

When you now start the server (and restart the client), you get the output shown in Figure 3-22, no matter if you used the [OneWay()] attribute or not. The interesting thing is that when using [OneWay()], the call to `SetValue()` finishes *before* the server completes the method. This is because in reality the client just *ignores* the server's response when using one-way method calls.

■ **Caution** Always remember that the client *ignores* the server's output and doesn't even check whether the server is running when using one-way methods!

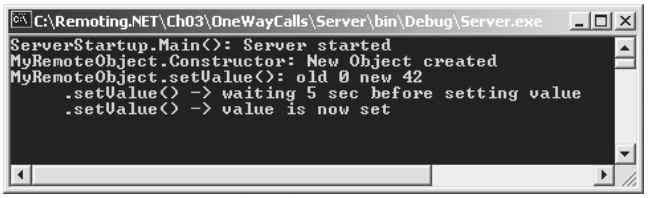
C:\Remoting.NET\Ch03\OneWayCalls\Server\bin\Debug\Server.exe
ServerStartup.Main(): Server started
MyRemoteObject.Constructor: New Object created
MyRemoteObject.setValue(): old 0 new 42
.setValue() -> waiting 5 sec before setting value
.setValue() -> value is now set

Figure 3-22. *Output on the server—independent of [OneWay()] attribute*

Multiserver Configuration

When using multiple servers in an application in which remote objects on one server will be passed as parameters to methods of a second server's object, there are a few things you need to consider.

Before talking about cross-server execution, I show you some details of remoting with MarshalByRefObjects. As the name implies, these objects are marshaled by reference—instead of passing a copy of the object over the network, only a pointer to this object, known as an ObjRef, will travel. Contrary to common pointers in languages like C++, ObjRefs don't reference a memory address but instead contain a network address (like a TCP/IP address and TCP port) and an object ID that's employed on the server to identify which object instance is used by the calling client. (You can read more on ObjRefs in Chapter 7.) On the client side these ObjRefs are encapsulated by a proxy object (actually, by two proxies, but you also get the chance to read more on those in Chapter 7).

After creating two references to client-activated objects on a remote server, for example, the client will hold two TransparentProxy objects. These objects will both contain an ObjRef object, which will in turn point to one of the two distinct CAOs. This is shown in Figure 3-23.

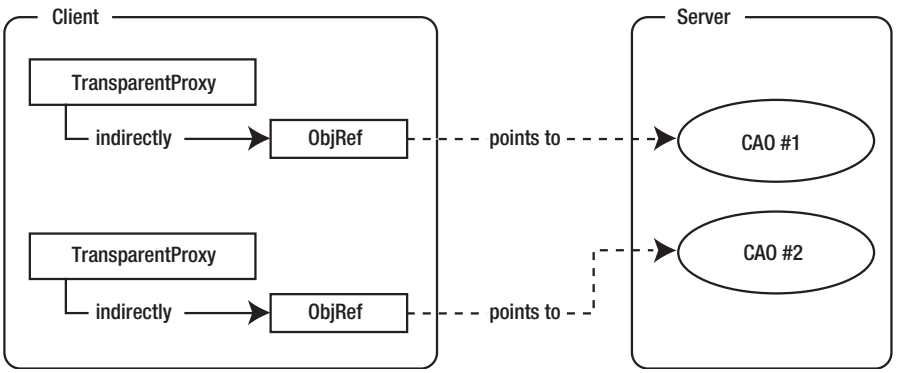

Figure 3-23. *ObjRefs are pointing to server-side objects.*

When a variable referencing a MarshalByRefObject is passed as a parameter to a remote function, the following happens: the ObjRef is taken from the proxy object, gets serialized (ObjRef is marked as [Serializable]), and is passed to the remote machine (the second server in this example). On this machine, new proxy objects are generated from the deserialized ObjRef. Any calls from the second machine to the remote object are placed directly on the first server without any intermediate steps via the client.

■ **Note** As the second server will contact the first one directly, there has to be a means of communication between them; that is, if there is a firewall separating the two machines, you have to configure it to allow connections from one server to the other.

Examining a Sample Multiserver Application

In the following example, I show you how to create a multiserver application in which Server 1 will provide a Singleton object that has an instance variable of type int. The client will obtain a remote reference to this object and pass it to a "worker object" located on a secondary server. This worker object is a SingleCall service providing a `DoSomething()` method, which takes an instance of the first object as a parameter. Figure 3-24 shows the Unified Modeling Language (UML) diagram for this setup.

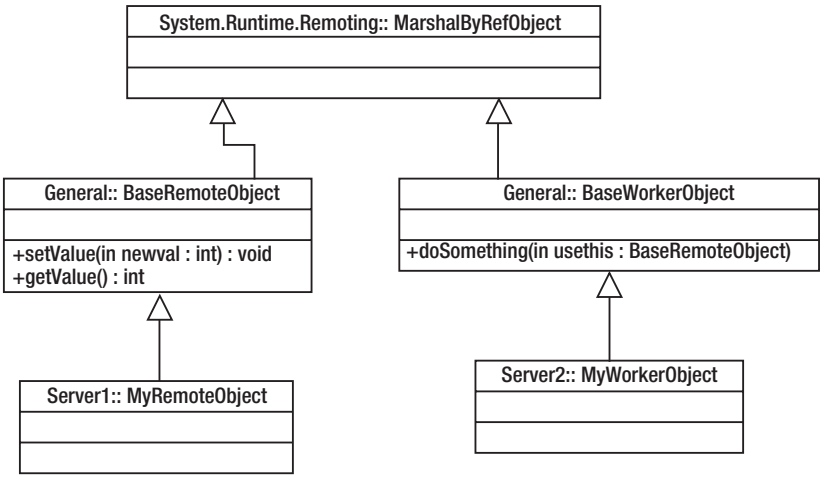

Figure 3-24. *UML diagram of the multiserver example*

■ **Note** If you use version 1.0 of the .NET Framework, you have to change the approach from using interfaces in General.dll to using abstract base classes. This first version of the .NET Remoting framework did not correctly serialize the interface hierarchy in the ObjRef, so these interface casts would not succeed. This problem has been fixed with version 1.1.

Figures 3-25 to 3-27 illustrate the data flow between the various components. In Figure 3-25, you see the situation after the first method call of the client on the first server object. The client holds a proxy object containing the ObjRef that points to the server-side Singleton object.

■ **Note** I use IDs like MRO#1 for an instance of MyRemoteObject not because that's .NET-like, but because it allows me to more easily refer to a certain object instance when describing the architecture.

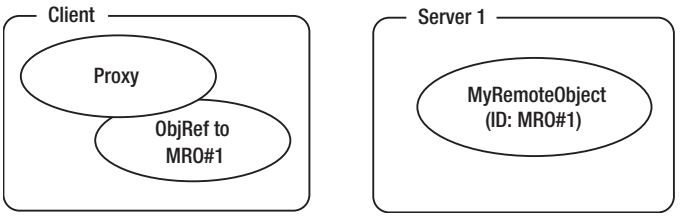

Figure 3-25. *Client and single server*

In the next step, which you can see in Figure 3-26, the client obtains a reference to the MarshalByRefObject called MyWorkerObject on the second server. It calls a method and passes its reference to the first server's object as a parameter. The ObjRef to this object (MRO#1) is serialized at the client and deserialized at the server, and a new proxy object is generated that sits on the second server and points to the object on the first (Figure 3-27). When MWO#1 now calls a method on MRO#1, the call will go directly from Server 2 to Server 1.

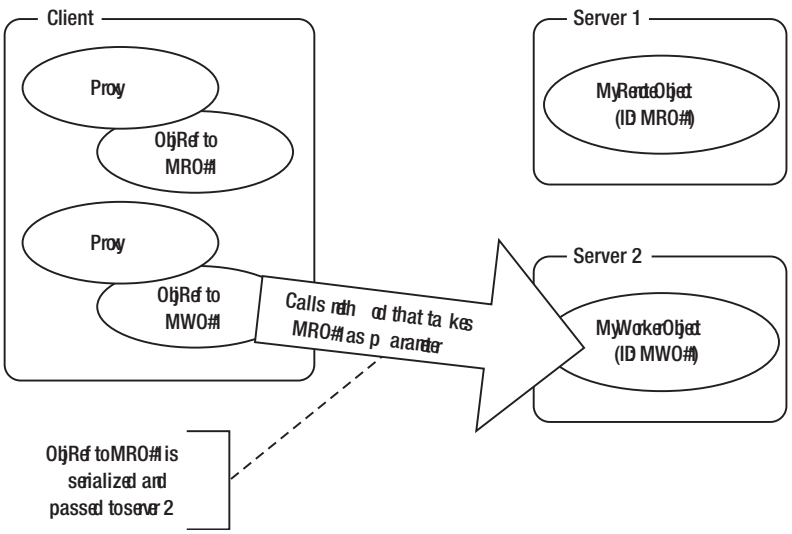

Figure 3-26. *Client calls a method on the second server with MRO#1 as parameter.*

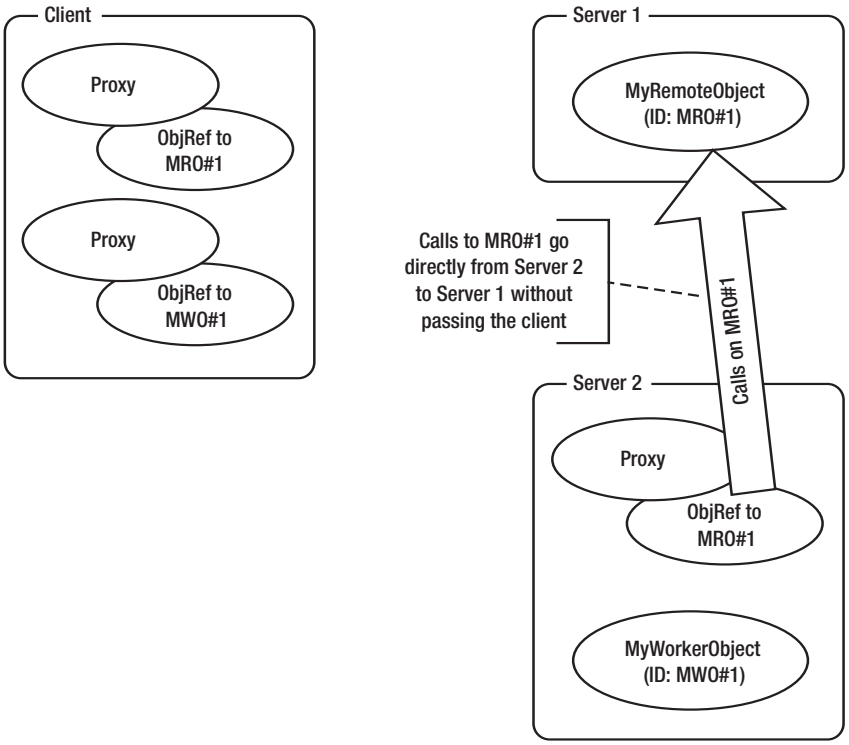

Figure 3-27. *Calls to the first server will go there directly without passing the client.*

Implementing the Shared Assembly

In the shared assembly, which is shown in Listing 3-19, I defined the two interfaces for the remote object and for the remote worker object.

Listing 3-19. *Defining the Two Interfaces in the Shared Assembly*

```
using System;

namespace General
{
   public interface IRemoteObject
   {
      void SetValue(int newval);
      int GetValue();
   }

   public interface IWorkerObject
   {
      void DoSomething(IRemoteObject usethis);
   }
}
```

The IRemoteObject's implementation is a Singleton located on the first server, and it allows the client to set and read an int as state information. The IWorkerObject's implementation is placed in Server 2 and provides a method that takes an object of type IRemoteObject as a parameter.

Implementing the First Server

The first server very closely resembles the servers from the other examples and is shown in Listing 3-20.

Listing 3-20. *The First Server*

```
using System;
using System.Runtime.Remoting;
using General;
using System.Runtime.Remoting.Channels.Http;
using System.Runtime.Remoting.Channels;

namespace Server
{
   class MyRemoteObject: MarshalByRefObject, IRemoteObject
   {
      int myvalue;

      public MyRemoteObject()
      {
         Console.WriteLine("MyRemoteObject.Constructor: New Object created");
      }

      public void SetValue(int newval)
      {
         Console.WriteLine("MyRemoteObject.SetValue(): old {0} new {1}",
                           myvalue,newval);
         myvalue = newval;
      }

      public int GetValue()
      {
         Console.WriteLine("MyRemoteObject.GetValue(): current {0}",myvalue);
         return myvalue;
      }
   }

   class ServerStartup
   {
      static void Main(string[] args)
      {
         Console.WriteLine ("ServerStartup.Main(): Server [1] started");
```

```
            HttpChannel chnl = new HttpChannel(1234);
            ChannelServices.RegisterChannel(chnl);
            RemotingConfiguration.RegisterWellKnownServiceType(
                  typeof(MyRemoteObject),
                  "MyRemoteObject.soap",
                  WellKnownObjectMode.Singleton);

            // the server will keep running until keypress.
            Console.ReadLine();
        }
    }
}
```

Implementing the Second Server

The second server works differently from those in prior examples. It provides a SingleCall object that accepts an IWorkerObject as a parameter. The SAO will contact this remote object, read and output its state, and change it before returning.

For the second server, you have to use a different startup code. The reason is that, starting with version 1.1 of the .NET Framework, you have to explicitly allow passing of remote references to your server application by changing the typeFilterLevel.[3] This second server is shown in Listing 3-21.

■ **Note** When running two servers on one machine, you have to give the servers different port numbers. Only one application can occupy a certain port at any given time. When developing production-quality applications, you should always allow the user or system administrator to configure the port numbers in a configuration file, via the registry or using a GUI.

■ **Note** When using version 1.0 of the .NET Framework, you can omit the additional construction information for the HttpChannel. This is only required starting with version 1.1.

Listing 3-21. *The Second Server*

```
using System;
using System.Runtime.Remoting;
using General;
using System.Runtime.Remoting.Channels.Http;
using System.Runtime.Remoting.Channels;
using System.Collections;
```

---

3. I'll talk more about this setting and the extended constructor for HttpChannel in Chapter 4.

```
namespace Server
{

   class MyWorkerObject: MarshalByRefObject, IWorkerObject
   {

      public MyWorkerObject()
      {
         Console.WriteLine("MyWorkerObject.Constructor: New Object created");
      }

      public void DoSomething(IRemoteObject usethis)
      {
         Console.WriteLine("MyWorkerObject.doSomething(): called");
         Console.WriteLine("MyWorkerObject.doSomething(): Will now call" +
                       "getValue() on the remote obj.");

         int tmp = usethis.GetValue();
         Console.WriteLine("MyWorkerObject.doSomething(): current value of " +
                        "the remote obj.; {0}", tmp);

         Console.WriteLine("MyWorkerObject.doSomething(): changing value to 70");
         usethis.SetValue(70);
      }
   }

   class ServerStartup
   {
      static void Main(string[] args)
      {
         Console.WriteLine ("ServerStartup.Main(): Server [2] started");
         SoapServerFormatterSinkProvider prov =
               new SoapServerFormatterSinkProvider();
         prov.TypeFilterLevel =
               System.Runtime.Serialization.Formatters.TypeFilterLevel.Full;

         IDictionary props = new Hashtable();
         props["port"] = 1235;

         HttpChannel chan = new HttpChannel(props, null, prov);
         ChannelServices.RegisterChannel( chan );

         RemotingConfiguration.RegisterWellKnownServiceType(
               typeof(MyWorkerObject),
               "MyWorkerObject.soap",
               WellKnownObjectMode.SingleCall);
```

```
            // the server will keep running until keypress.
            Console.ReadLine();
        }
    }
}
```

Running the Sample

When the client is started, it first acquires a remote reference to MyRemoteObject running on the first server. It then changes the object's state to contain the value 42 and afterwards reads the value from the server and outputs it in the console window (see Figure 3-28).

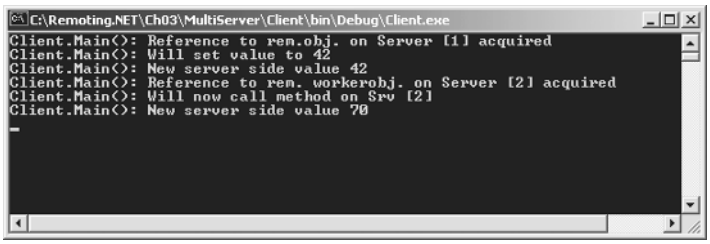
C:\Remoting.NET\Ch03\MultiServer\Client\bin\Debug\Client.exe
Client.Main(): Reference to rem.obj. on Server [1] acquired
Client.Main(): Will set value to 42
Client.Main(): New server side value 42
Client.Main(): Reference to rem. workerobj. on Server [2] acquired
Client.Main(): Will now call method on Srv [2]
Client.Main(): New server side value 70

Figure 3-28. *The client's output*

Next it fetches a remote reference to MyWorkerObject running on the second server. The client calls the method `DoSomething()` and passes its reference to MyRemoteObject as a parameter. When Server 2 receives this call, it contacts Server 1 to read the current value from MyRemoteObject and afterwards changes it to 70. (See Figures 3-29 and 3-30.)

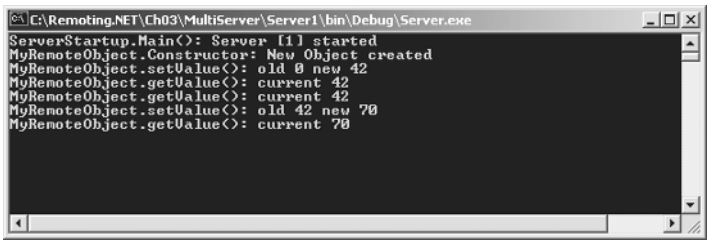

Figure 3-29. *The first server's output*

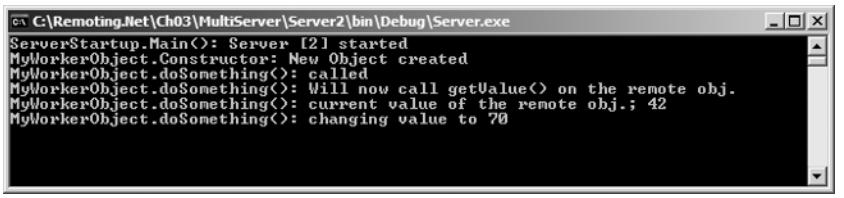
C:\Remoting.Net\Ch03\MultiServer\Server2\bin\Debug\Server.exe
ServerStartup.Main(): Server [2] started
MyWorkerObject.Constructor: New Object created
MyWorkerObject.doSomething(): called
MyWorkerObject.doSomething(): Will now call getValue() on the remote obj.
MyWorkerObject.doSomething(): current value of the remote obj.; 42
MyWorkerObject.doSomething(): changing value to 70

Figure 3-30. *The second server's output*

When the call from the client to the second server returns, the client again contacts MyRemoteObject to obtain the current value, 70, which shows that your client really has been talking to the same object from both processes.

Sharing Assemblies

As you've seen in this chapter, .NET Remoting applications need to share common information about remoteable types between server and client. Contrary to other remoting schemas like CORBA, Java RMI, and J2EE EJBs, with which you don't have a lot of choice for writing these shared interfaces, base classes, and metadata, the .NET Framework gives you at least four possible ways to do so, as I discuss in the following sections.

Shared Implementation

The first way to share information about remoteable types is to implement your server-side objects in a shared assembly and deploy this to the client as well. The main advantage here is that you don't have any extra work. Even though this might save you some time during implementation, I really recommend against this approach. Not only does it violate the core principles of distributed application development, but it also allows your clients, which are probably third parties accessing your ERP system to automate order entry, to use ILDASM or one of the upcoming MSIL-to-C# decompilers to disassemble and view your business logic. Unfortunately, this approach is shown in several MSDN examples.

Nevertheless, there are application scenarios that depend on this way of sharing the metadata. When you have an application that can be used either connected or disconnected and will access the same logic in both cases, this might be the way to go. You can then "switch" dynamically between using the local implementation and using the remote one.

Shared Interfaces

In the first examples in this book, I show the use of shared interfaces. With this approach, you create an assembly that is copied to both the server and the client. The assembly contains the interfaces that will be implemented by the server.

■ **Tip** I absolutely recommend this approach for most .NET Remoting applications!

Shared Base Classes

Instead of sharing interfaces between the client and the server, you can also create abstract base classes in a shared assembly. The server-side object will inherit from these classes and implement the necessary functionality. I recommend against this approach because it pollutes the inheritance hierarchy with too many protocol- and framework-specific constructs.

SoapSuds-Generated Metadata

Another approach is to use SoapSuds.exe to extract the metadata (that is, the type definition) from a running server or an implementation assembly and generate a new assembly that contains only this meta information. You will then be able to reference this assembly in the client application without manually generating any intermediate shared assemblies.

■ **Caution** Even though this approach seems intriguing at first glance, I recommend against using it for most general .NET Remoting applications. SoapSuds is a tool that has been designed earlier in the .NET lifetime, and has been superceded by development since. If you want to work with Web Services, WSDL, and SOAP, then you should really use ASP.NET instead of .NET Remoting. **I will nevertheless show how to use SoapSuds in the remainder of this chapter, but this should not be interpreted as a recommendation.**

Calling SoapSuds

SoapSuds is a command-line utility, therefore the easiest way to start it is to bring up the Visual Studio .NET Command Prompt by selecting Start ➤ Programs ➤Microsoft Visual Studio .NET 2003 ➤ Visual Studio .NET Tools. This command prompt will have the path correctly set so that you can execute all .NET Framework SDK tools from any directory.

Starting SoapSuds without any parameters will give you detailed usage information. To generate a metadata DLL from a running server, you have to call SoapSuds with the -url parameter.

```
soapsuds -url:<URL> -oa:<OUTPUTFILE>.DLL -nowp
```

■ **Note** You normally have to append ?wsdl to the URL your server registered for a SOA to allow SoapSuds to extract the metadata.

To let SoapSuds extract the information from a compiled DLL, you use the -ia parameter.

```
soapsuds -ia:<assembly> -oa:<OUTPUTFILE>.DLL -nowp
```

Wrapped Proxies

When you run SoapSuds in its default configuration (without the -nowp parameter) by passing only a URL as an input parameter and telling it to generate an assembly, it will create what is called a wrapped proxy. The wrapped proxy can only be used on SOAP channels and will directly store the path to your server. Normally you do not want this behavior, apart from some rare testing scenarios, in which you'd just like to quickly call a method on a server during development.

Implementing the Server

The server in this example will be implemented without any up-front definition of interfaces. You only need to create a simplistic SAO and register an HTTP channel to allow access to the metadata and the server-side object, as shown in Listing 3-22.

Listing 3-22. *Server That Presents a SAO*

```
using System;
using System.Runtime.Remoting;
using System.Runtime.Remoting.Channels.Http;
using System.Runtime.Remoting.Channels;

namespace Server
{

   class SomeRemoteObject: MarshalByRefObject
   {
      public void DoSomething()
      {
         Console.WriteLine("SomeRemoteObject.doSomething() called");
      }
   }

   class ServerStartup
   {
      static void Main(string[] args)
      {
         Console.WriteLine ("ServerStartup.Main(): Server started");

         HttpChannel chnl = new HttpChannel(1234);
         ChannelServices.RegisterChannel(chnl);
         RemotingConfiguration.RegisterWellKnownServiceType(
               typeof(SomeRemoteObject),
               "SomeRemoteObject.soap",
               WellKnownObjectMode.SingleCall);

         // the server will keep running until keypress.
         Console.ReadLine();
      }
   }
}
```

Generating the SoapSuds Wrapped Proxy

To generate a wrapped proxy assembly, use the SoapSuds command line shown in Figure 3-31. The resulting meta.dll should be copied to the client directory, as you will have to reference it when building the client-side application.

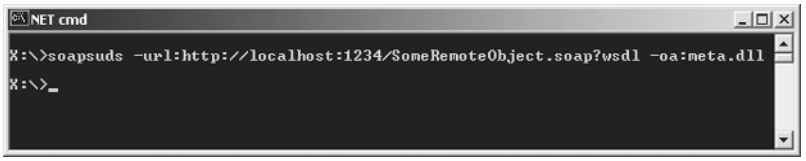

Figure 3-31. *SoapSuds command line used to generate a wrapped proxy*

Implementing the Client

Assuming you now want to implement the client application, you first have to set a reference to the meta.dll in the project's References dialog box in VS .NET. You can then use the Server namespace and directly instantiate a SomeRemoteObject using the `new` operator, as shown in Listing 3-23.

Listing 3-23. *Wrapped Proxies Simplify the Client's Source Code*

```
using System;
using Server;

namespace Client
{
   class Client
   {
      static void Main(string[] args)
      {
         Console.WriteLine("Client.Main(): creating rem. reference");
         SomeRemoteObject obj = new SomeRemoteObject();
         Console.WriteLine("Client.Main(): calling DoSomething()");
         obj.DoSomething();
         Console.WriteLine("Client.Main(): done ");

         Console.ReadLine();
      }
   }
}
```

Even though this code looks intriguingly simple, I recommend against using a wrapped proxy for several reasons: the server's URL is hard coded, and you can only use an HTTP channel and not a TCP channel.

When you start this client, it will generate the output shown in Figure 3-32. Check the server's output in Figure 3-33 to see that `DoSomething()` has really been called on the server-side object.

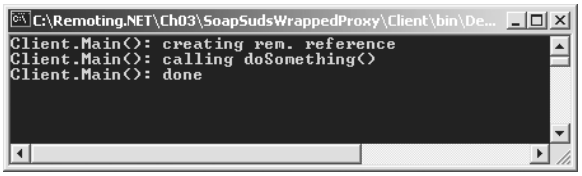

Figure 3-32. *Client's output when using a wrapped proxy*

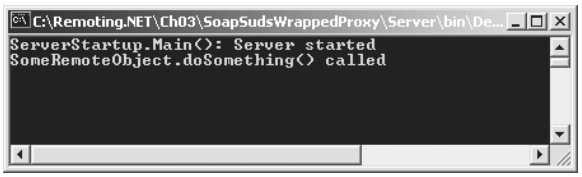

Figure 3-33. *The server's output shows that doSomething() has been called..*

Wrapped Proxy Internals

Starting SoapSuds with the parameter -gc instead of -oa:<assemblyname> will generate C# ccode in the current directory. You can use this code to manually compile a DLL or include it directly in your project.

Looking at the code in Listing 3-24 quickly reveals why you can use it without any further registration of channels or objects. (I stripped the SoapType attribute, which would normally contain additional information on how to remotely call the object's methods.)

Listing 3-24. *A SoapSuds-Generated Wrapped Proxy*

```
using System;
using System.Runtime.Remoting.Messaging;
using System.Runtime.Remoting.Metadata;
using System.Runtime.Remoting.Metadata.W3cXsd2001;

namespace Server {

    public class SomeRemoteObject :
            System.Runtime.Remoting.Services.RemotingClientProxy
    {
        // Constructor
        public SomeRemoteObject()
        {
            base.ConfigureProxy(this.GetType(),
                 "http://localhost:1234/SomeRemoteObject.soap");
        }

        public Object RemotingReference
        {
            get{return(_tp);}
        }
```

```
[SoapMethod(SoapAction="http://schemas.microsoft.com/clr/nsassem/
Server.SomeRemoteObject/Server#doSomething")]
        public void DoSomething()
        {
            ((SomeRemoteObject) _tp).doSomething();
        }

    }
}
```

What this wrapped proxy does behind the scenes is provide a custom implementation/extension of RealProxy (which is the base for RemotingClientProxy) so that it can be used transparently. This architecture is shown in detail in Chapter 7.

Nonwrapped Proxy Metadata

SoapSuds also allows the generation of nonwrapped proxy metadata as well. In this case, it will only generate empty class definitions, which can then be used by the underlying .NET Remoting TransparentProxy to generate the true method calls—no matter which channel you are using.

This approach also gives you the advantage of being able to use configuration files for channels, objects, and the corresponding URLs (more on this in the next chapter) so that you don't have to hard code this information. In the following example, you can use the same server as in the previous one, only changing the SoapSuds command and implementing the client in a different way.

Generating the Metadata with SoapSuds

As you want to generate a metadata-only assembly, you have to pass the -nowp parameter to SoapSuds to keep it from generating a wrapped proxy (see Figure 3-34).

Figure 3-34. *SoapSuds command line for a metadata-only assembly*

Implementing the Client

When using metadata-only output from SoapSuds, the client looks a lot different from the previous one. In fact, it closely resembles the examples I show you at the beginning of this chapter.

First you have to set a reference to the newly generated meta.dll from the current SoapSuds invocation and indicate that your client will be using this namespace. You can then proceed with the standard approach of creating and registering a channel and calling `Activator.GetObject()` to create a reference to the remote object. This is shown in Listing 3-25.

Listing 3-25. *The Client with a Nonwrapped Proxy*

```
using System;
using System.Runtime.Remoting;
using System.Runtime.Remoting.Channels.Http;
using System.Runtime.Remoting.Channels;
using Server;

namespace Client
{
   class Client
   {
      static void Main(string[] args)
      {
         HttpChannel chnl = new HttpChannel();
         ChannelServices.RegisterChannel(chnl);

         Console.WriteLine("Client.Main(): creating rem. reference");
         SomeRemoteObject obj = (SomeRemoteObject) Activator.GetObject (
            typeof(SomeRemoteObject),
            "http://localhost:1234/SomeRemoteObject.soap");

         Console.WriteLine("Client.Main(): calling doSomething()");
         obj.DoSomething();

         Console.WriteLine("Client.Main(): done ");
         Console.ReadLine();
      }
   }
}
```

When this client is started, both the client-side and the server-side output will be the same as in the previous example (see Figures 3-35 and 3-36).

Figure 3-35. *The Client's output when using a metadata-only assembly*

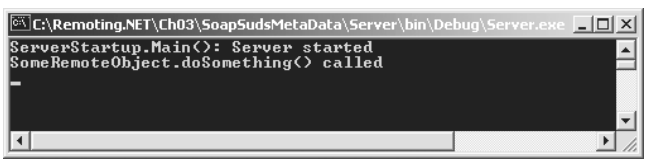

Figure 3-36. *The Server's output is the same as in the previous example.*

Summary

In this chapter you read about the basics of distributed .NET applications using .NET Remoting. You now know the difference between ByValue objects and MarshalByRefObjects, which can be either server-activated objects (SAO) or client-activated objects (CAO). You can call remote methods asynchronously, and you know about the dangers and benefits of one-way methods. You also learned about the different ways in which a client can receive the necessary metadata to access remote objects, and that you should normally use shared interfaces as a best practices approach.

It seems that the only thing that can keep you from developing your first real-world .NET Remoting application is that you don't yet know about various issues surrounding configuration and deployment of such applications. These two topics are covered in the following chapter.

CHAPTER 4

Configuration and Deployment

This chapter introduces you to the aspects of configuration and deployment of .NET Remoting applications. It shows you how to use configuration files to avoid the hard coding of URLs or channel information for your remote object.

You also learn about hosting your server-side components in Windows services or Internet Information Server (IIS)—the latter of which gives you the possibilities to deploy your components for authenticated or encrypted connections, which is covered in detail in Chapter 5.

To configure your applications, you can choose to either implement all channel and object registrations on your own or employ the standard .NET Remoting configuration files. These files are XML documents, and they allow you to configure nearly every aspect of remoting, ranging from various default and custom channels (including custom properties for your own channels) to the instantiation behavior of your objects.

The big advantage, and the main reason you should always use configuration files in production applications, is that you can change the application's remoting behavior without the need to recompile. For example, you could create a configuration file for users located directly within your LAN who might use a direct TCP channel, and another file for WAN users who will use a secured HTTP channel with SSL encryption. All this can be done without changing a single line in your application's source code.

Note Of course, this also adds the possibility for the user to change your configuration file. If this is not desired, you should make sure to protect the configuration file with security Access Control Lists (ACL) to allow only administrators to edit the file.

With other remoting architectures, the choice of deployment and configuration is largely determined when choosing the framework. With Java EJBs, for example, the "container," which can be compared to an application server, defines the means of configuration and locks you into a single means of deployment. The same is true for the COM+ component, which has to be hosted in Windows Component services.

In the .NET Remoting framework, you have several possibilities for operation: you can run your remote objects in a distinct stand-alone application (as shown in the previous examples in this book), run them in a Windows service, or host them in IIS.

Configuration Files

.NET Remoting configuration files allow you to specify parameters for most aspects of the remoting framework. These files can define tasks as simple as registering a channel and specifying a type as a server-activated object, or can be as complex as defining a whole chain of IMessageSinks with custom properties.

Instead of writing code like this on the server:

```
HttpChannel chnl = new HttpChannel(1234);
ChannelServices.RegisterChannel(chnl);
RemotingConfiguration.RegisterWellKnownServiceType(
               typeof(CustomerManager),
               "CustomerManager.soap",
               WellKnownObjectMode.Singleton);
```

You can use a configuration file that contains the following XML document to specify the same behavior:

```
<configuration>
  <system.runtime.remoting>
    <application>
      <channels>
        <channel ref="http" port="1234" />
      </channels>
      <service>
        <wellknown mode="Singleton"
                   type="Server.CustomerManager, Server"
                   objectUri="CustomerManager.soap" />
      </service>

    </application>
  </system.runtime.remoting>
</configuration>
```

To employ this configuration file in your application, you have to call `RemotingConfiguration.Configure()` and pass the filename of your *.config file to it.

```
String filename = "server.exe.config";
RemotingConfiguration.Configure(filename);
```

■ **Note** As a convention for .NET applications, the configuration filename should be <applicationname>.config, whereas an application filename includes the extension .exe or .dll. Visual Studio automates this process for you, so that you can simply add a file called app.config to your solution. Upon deployment, this file will be renamed according to the schema presented previously, and will be copied to the appropriate output directory (debug/release).

Watch for the Metadata!

Instead of using `Activator.GetObject()` and passing a URL to it, you can use the `new` operator after loading the configuration file with `RemotingConfiguration.Configure()`.

In terms of the sample application in Chapter 2, this means that instead of the following call:

```
ICustomerManager mgr = (ICustomerManager) Activator.GetObject(
         typeof(ICustomerManager),
         "http://localhost:1234/CustomerManager.soap");
```

you might simply use this statement after the configuration file has been loaded:

```
CustomerManager mgr = new CustomerManager()
```

And here the problem starts: you need the definition of the class CustomerManager on the client. The interface is not sufficient anymore, because you cannot use `IInterface x = new IInterface()`, as this would represent the instantiation of an interface, which is not possible.

In Chapter 3, I showed you several tools for supplying the necessary metadata in a shared assembly: interfaces, abstract base classes, and SoapSuds-generated metadata-only assemblies. When using the configured `new` operator, you won't be able to employ abstract base classes or interfaces—instead you basically have to resort to shipping the implementation or using SoapSuds-generated metadata.

■ **Note** I will show you an alternative approach later in this chapter which allows you to take advantage of configuration files with interface-based remote objects.

The Problem with SoapSuds

When your application includes only SAOs/CAOs (and no [Serializable] objects), you're usually fine with using `soapsuds -ia:<assembly> -nowp -oa:<meta_data.dll>` to generate the necessary metadata. However, when you are using [Serializable] objects, which not only hold some data but also have methods defined, you need to provide their implementation (the General.dll in the examples) to the client as well.

To see the problem and its solution, take a look at Listing 4-1. This code shows you a [Serializable] class in a shared assembly that will be called General.dll.

Listing 4-1. *A Shared [Serializable] Class*

```
using System;
namespace General
{

   [Serializable]
   public class Customer
   {
      public String FirstName;
      public String LastName;
      public DateTime DateOfBirth;
```

```
      public int GetAge()
      {
         TimeSpan tmp = DateTime.Today.Subtract(DateOfBirth);
         return tmp.Days / 365; // rough estimation
      }
   }
}
```

On the server side you use the following configuration file, which allows you to write `CustomerManager obj = new CustomerManager()` to acquire a reference to the remote object.

```
<configuration>
  <system.runtime.remoting>
    <application>
      <channels>
        <channel ref="http" port="1234" />
      </channels>
      <service>
        <wellknown mode="Singleton"
                   type="Server.CustomerManager, Server"
                   objectUri="CustomerManager.soap" />
      </service>

    </application>
  </system.runtime.remoting>
</configuration>
```

The server itself, which is shown in Listing 4-2, implements a MarshalByRefObject that provides a `GetCustomer()` method, which will return a Customer object by value.

Listing 4-2. *The Server-Side Implementation of CustomerManager*

```
using System;
using System.Runtime.Remoting;
using General;

namespace Server
{

   class CustomerManager: MarshalByRefObject
   {
      public Customer GetCustomer(int id)
      {
         Customer tmp = new Customer();
         tmp.FirstName = "John";
```

```
            tmp.LastName = "Doe";
            tmp.DateOfBirth = new DateTime(1970,7,4);
            return tmp;
         }
      }

      class ServerStartup
      {
         static void Main(string[] args)
         {
            Console.WriteLine ("ServerStartup.Main(): Server started");

            String filename = "server.exe.config";
            RemotingConfiguration.Configure(filename);

            // the server will keep running until keypress.
            Console.WriteLine("Server is running, Press <return> to exit.");
            Console.ReadLine();
         }
      }
}
```

After compiling, starting the server, and running the SoapSuds command shown in Figure 4-1, you'll unfortunately end up with a little bit more output than expected.

Figure 4-1. *SoapSuds command line for extracting the metadata*

The Generated_General.dll file (which you can see in Figure 4-2) contains not only the metadata for Server.CustomerManager, but also the definition for General.Customer. This is because SoapSuds generates metadata-only assemblies for all classes and assemblies that are referenced by your server. You'll run into the same problems when referencing parts from mscorlib.dll or other classes from the System.* namespaces.

Comparing Generated_General.dll to the original General.Customer in General.dll (the one that has been created by compiling the shared project) in Figure 4-3, you can see that although the generated Customer class contains all defined fields, it does not include the `GetAge()` method.

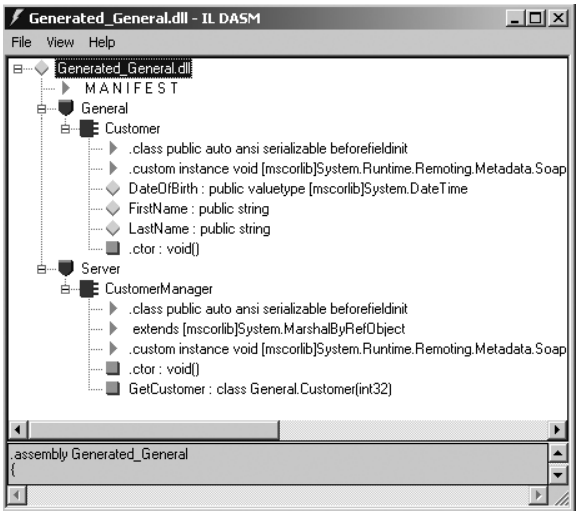

Figure 4-2. *The Generated_General.dll that has been created by SoapSuds*

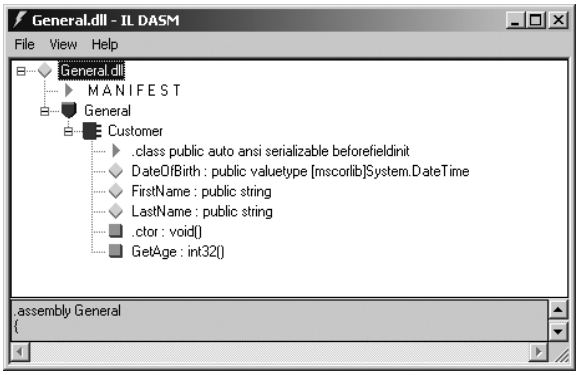

Figure 4-3. *The original General.dll contains the method Customer.GetAge().*

You can now safely assume that using the Generated_General.dll will not be sufficient for the application; after all, you *want* access to the `GetAge()` method.

If you try to reference both General.dll and Generated_General.dll in your client application, you will end up with a namespace clash. Both assemblies contain the same namespace and the same class (General.Customer). Depending on the order of referencing the two DLLs, you'll end up with either a compile-time error message telling you that the `GetAge()` method is missing *or* an InvalidCastException when calling `CustomerManager.GetCustomer()`.

Using SoapSuds to Generate Source Code

There is, however, a possibility to work around this problem: although SoapSuds can be used to generate DLLs from the WSDL information provided by your server, it can also generate C# code files, including not only the definition but also the required SoapMethodAttributes, so that the remoting framework will know the server's namespace identifier.

To generate code instead of DLLs, you have to specify the parameter -gc instead of -oa:<someassembly>.dll. This command, shown in Figure 4-4, generates one C# source code file for each server-side assembly. You will therefore end up with one file called general.cs and another called server.cs (both are placed in the current directory).

Figure 4-4. *SoapSuds command line for generating C# code*

The generated server.cs file is shown in Listing 4-3.

Listing 4-3. *The SoapSuds-Generated Server.cs File*

```
using System;
using System.Runtime.Remoting.Messaging;
using System.Runtime.Remoting.Metadata;
using System.Runtime.Remoting.Metadata.W3cXsd2001;
namespace Server {

    [Serializable, SoapType(XmlNamespace="http://schemas.microsoft.com/clr/n
sassem/Server/Server%2C%20Version%3D1.0.678.38058%2C%20Culture%3Dneutral%2C%
20PublicKeyToken%3Dnull", XmlTypeNamespace="http://schemas.microsoft.com/clr
/nsassem/Server/Server%2C%20Version%3D1.0.678.38058%2C%20Culture%3Dneutral%2
C%20PublicKeyToken%3Dnull")]
    public class CustomerManager : System.MarshalByRefObject
    {
        [SoapMethod(SoapAction="http://schemas.microsoft.com/clr/nsassem/Ser
ver.CustomerManager/Server#GetCustomer")]
        public General.Customer GetCustomer(Int32 id)
        {
            return((General.Customer) (Object) null);
        }
    }
}
```

Generally speaking, these lines represent the interface and the attributes necessary to clearly resolve this remoting call to the server.

Instead of including the Generated_General.dll file (which also contains the namespace General), you can include this C# file in the client-side project or compile it to a DLL in a separate C# project.

You can now safely reference the shared General.dll, without any namespace clashes, in your project to have access to the Customer class's implementation.

Porting the Sample to Use Configuration Files

■ **Note** I will use SoapSuds-generated metadata for the following introduction to configuration files. The use of SoapSuds for your real production applications is not a best practice and should be avoided whenever possible. I will show you an alternative approach in the section "What About Interfaces?" later in this chapter that allows you to use configuration files with interface-based remote objects.

Taking the first sample application in Chapter 2 (the CustomerManager SAO that returns a Customer object by value), I'll show you here how to enhance it to use configuration files.

Assume that on the server side of this application you want an HTTP channel to listen on port 1234 and provide remote access to a well-known Singleton object. You can do this with the following configuration file:

```
<configuration>
  <system.runtime.remoting>
    <application>

      <channels>
        <channel ref="http" port="1234" />
      </channels>

      <service>
        <wellknown mode="Singleton"
                   type="Server.CustomerManager, Server"
                   objectUri="CustomerManager.soap" />
      </service>

    </application>
  </system.runtime.remoting>
</configuration>
```

The server-side implementation will simply load the configuration file and wait for <Return> to be pressed before exiting the application. The implementation of CustomerManager is the same as shown previously, and only the server's startup code is reproduced here:

```
using System;
using System.Runtime.Remoting;
using General;

namespace Server
{
   class ServerStartup
   {
      static void Main(string[] args)
      {
```

```
            String filename = "server.exe.config";
            RemotingConfiguration.Configure(filename);

            Console.WriteLine("Server is running. Press <Return> to exit.");
            Console.ReadLine();
        }
    }
}
```

Creating the Client

The client will consist of two source files, one of them being the previously mentioned SoapSuds-generated server.cs and the second will contain the real client's implementation. It will also have a reference to the shared General.dll.

To allow the client to access the server-side Singleton, you can use the following configuration file to avoid hard coding of URLs:

```
<configuration>
  <system.runtime.remoting>
    <application>

      <client>
        <wellknown type="Server.CustomerManager, Client"
                   url="http://localhost:1234/CustomerManager.soap" />
      </client>

    </application>
  </system.runtime.remoting>
</configuration>
```

Even before I get to the details of configuration files, I want to mention what the attribute "type" in the XML tag <wellknown> contains. In the previous example, you can see that on the server side the tag includes "Server.CustomerManager, Server" and on the client-side it includes "Server.CustomerManager, Client".

The format is generally "<namespace>.<class>, <assembly>", so when a call to the remote object is received at the server, it will create the class Server.CustomerManager from its Server assembly.

■ **Caution** Make sure you do *not* include the .dll or .exe extension here as the .NET Remoting framework will not give you any error messages in this case. Instead, your application just won't work as expected. If you want to be sure that you're dealing with a remote object, you can call `RemotingServices.IsTransparentProxy()` right after the creation of the remote reference.

On the client side the format is a little bit different. The preceding entry, translated into plain English, more or less reads, "If someone creates an instance of the class Server.CustomerManager, which is located in the Client assembly, then generate a remote reference pointing to

`http://localhost:1234/CustomerManager.soap`." You therefore have to specify the client's assembly name in the type attribute of this tag. This differs when using a SoapSuds-generated DLL, in which case you would have to include the name of this generated assembly there.

■ **Caution** When a typo occurs in your configuration file, such as when you misspell the assembly name or the class name, there won't be an exception during "x = new Something()". Instead, you will get a reference to the *local* object. When you subsequently call methods on it, they will return *null*.

The C# code of the client application (excluding the SoapSuds-generated server.cs) is shown in Listing 4-4.

Listing 4-4. *The Working Client Application (Excluding server.cs)*

```
using System;
using System.Runtime.Remoting;
using General;  // from General.DLL
using Server; // from server.cs

namespace Client
{
   class Client
   {
      static void Main(string[] args)
      {
         String filename = "client.exe.config";
         RemotingConfiguration.Configure(filename);

         CustomerManager mgr = new CustomerManager();

         Console.WriteLine("Client.Main(): Reference to CustomerManager" +
                           " acquired");

         Customer cust = mgr.getCustomer(4711);
         int age = cust.getAge();
         Console.WriteLine("Client.Main(): Customer {0} {1} is {2} years old.",
            cust.FirstName,
            cust.LastName,
            age);

         Console.ReadLine();
      }
   }
}
```

When server and client are started, you will see the familiar output shown in Figures 4-5 and 4-6.

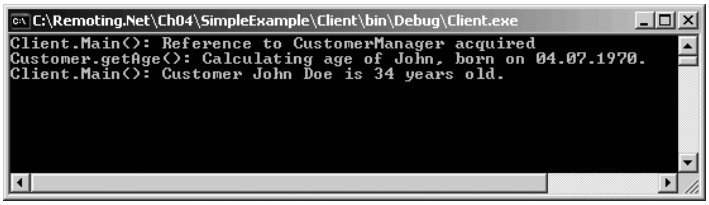
C:\Remoting.Net\Ch04\SimpleExample\Client\bin\Debug\Client.exe
Client.Main(): Reference to CustomerManager acquired
Customer.getAge(): Calculating age of John, born on 04.07.1970.
Client.Main(): Customer John Doe is 34 years old.

Figure 4-5. *Client's output when using the configuration file*

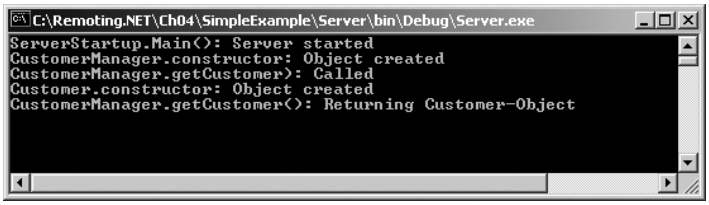

Figure 4-6. *Server's output when using the configuration file*

Standard Configuration Options

All .NET configuration files start with <configuration>, and this applies to remoting configuration files as well.

A remoting configuration file basically contains the following structure:

```
<configuration>
   <system.runtime.remoting>
      <application>
         <lifetime />
         <channels />
         <service />
         <client />
      </application>
      <channels />
      <channelSinkProviders />
      <debug />
   </system.runtime.remoting>
</configuration>
```

General Configuration Options

Besides the <application> configuration, which is used for our specific clients and services, there are some general configuration options listed in the following table:

Option	Description
channels	Contains channel templates that can be used in the application's channel options using the ref attribute as you will see it later in this chapter. This attribute is used especially when you create your own channels (see Chapter 14).
channelSinkProviders	Contains a list of providers of channel sinks that can be inserted in the channel sink chain. More information about channel sinks can be found in Chapter 12.
debug	The debug option tells the .NET Remoting infrastructure to load all types immediately on application startup. That makes it easier catching typing errors in the configuration file as errors occur immediately after startup and not only when types are instantiated.

The <debug> element really helps finding errors in configuration early. In my opinion the best example is misconfiguration of the server. Let's just take the first example of this chapter where you saw the simple example from the first chapter modified to make use of configuration files. Let's modify the server's configuration file like the following:

```
<configuration>
  <system.runtime.remoting>
    <application>
      <channels>
        <channel ref="http" port="1234" />
      </channels>
      <service>
        <wellknown mode="Singleton"
                   type="Server.CustomerManager, ServerMissConfig"
                   objectUri="CustomerManager.soap" />
      </service>
    </application>
  </system.runtime.remoting>
</configuration>
```

Actually, the assembly *ServerMissConfig* doesn't exist anywhere. But when starting, the server configuration succeeds with no errors. But when will you see the error? The client gets the exception when trying to call the server the first time as you can see in Figure 4-7. The reason is that the server will by default only check for the existence of a given type as soon as a request is handled.

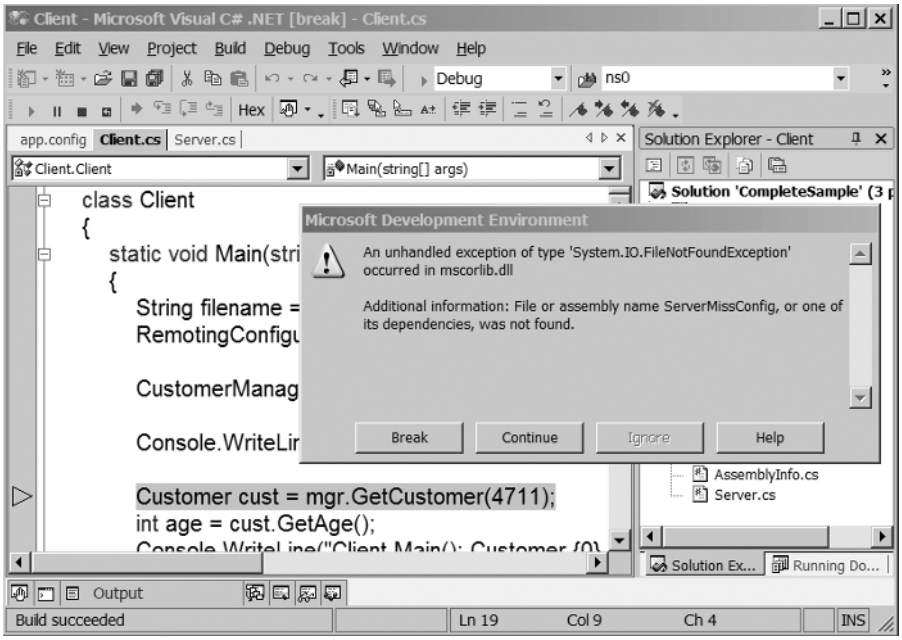

Figure 4-7. *The error occurs on the client side although server config is wrong.*

With the next step, let's add the <debug> configuration option as shown in the following code snippet:

```
<configuration>
  <system.runtime.remoting>
    <application>
      <channels>
        <channel ref="http" port="1234" />
      </channels>
      <service>
        <wellknown mode="Singleton"
                   type="Server.CustomerManager, ServerMissConfig"
                   objectUri="CustomerManager.soap" />
      </service>
    </application>
    <debug loadTypes="true" />
  </system.runtime.remoting>
</configuration>
```

This time the exception is thrown at the right position on the server, as you can see in Figure 4-8.

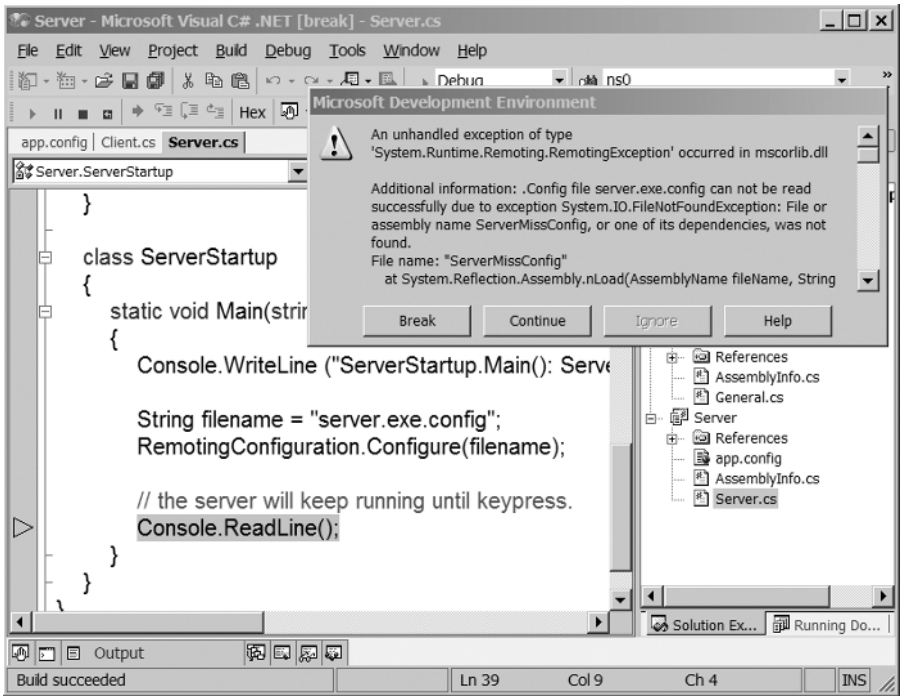

Figure 4-8. *The error now thrown after RemotingConfiguration.Configure()*

Lifetime

Use the <lifetime> tag to configure your object's default lifetime (as discussed in Chapter 3). Valid attributes for the <lifetime> tag are listed here:

Attribute	Description
leaseTime	The initial time to live (TTL) for your objects (default is 5 minutes)
sponsorshipTimeout	The time to wait for a sponsor's reply (default is 2 minutes)
renewOnCallTime	The time to add to an object's TTL when a method is called (default is 2 minutes)
leaseManagerPollTime	The interval in which your object's TTL will be checked whether they reached zero (default is 10 seconds)

All attributes are optional and may be specified in different time units. Valid units are D for days, H for hours, M for minutes, S for seconds, and MS for milliseconds. When no unit is specified, the system will default to S. Combinations such as 1H5M are *not* supported.

Here is an example for a very short-lived object:

```
<lifetime
   leaseTime="90MS"
   renewOnCallTime="90MS"
   leaseManagerPollTime="100MS"
/>
```

Channels

The <channels> tag contains one or more channel entries. It only serves as a collection for these and doesn't have any XML attributes assigned to it.

To register a server-side TCP channel that listens on port 1234, you can specify the following configuration section:

```
<channels>
   <channel ref="tcp" port="1234">
</channels>
```

Channel

The <channel> tag allows you to specify a port number for the server application, to reference custom channels, and to set additional attributes on channels. When you want to use the default HTTP channel or TCP channel, this tag does not have to be specified on the client because these channels will be registered automatically by the framework. On the server, you have to specify at least a port number on which the server-side channel will listen.

You have basically two ways of referencing channels: using a named reference for a predeclared channel or specifying the exact type (namespace, class name, and assembly—and version information, if the assembly is in the GAC) for the channel's implementation. Valid attributes for the <channel> tag are as follows:

Attribute	Description
ref	Reference for a predefined channel ("tcp" or "http") or reference to a channel that has been defined in a configuration file.
displayName	Attribute only used for the .NET Framework Configuration Tool.
type	Attribute that is mandatory when ref has not been specified. Contains the exact type (namespace, class name, assembly) of the channel's implementation. When the assembly is in the GAC, you have to specify version, culture, and public key information as well. For an example of this, see the default definition of HTTP channel in your machine.conf file (which is located in %WINDIR%\Microsoft.NET\Framework\<Framework_Version>\CONFIG).
port	Server-side port number. When using this attribute on a client, 0 should be specified if you want your client-side objects to be able to receive callbacks from the server.

In addition to the preceding configuration properties, each channel type can have other configuration information specific to itself. For the HTTP channel, these are as follows:

Attribute	Description
name	Name of the channel (default is "http"). When registering more than one channel, these names have to be unique or an empty string ("") has to be specified. The value of this attribute can be used when calling ChannelServices.GetChannel().
priority	Indicator of the likelihood for this channel to be chosen by the framework to transfer data (default is 1). The higher the integer, the greater the possibility. Negative numbers are allowed.

Continues

Attribute	Description
clientConnectionLimit	Number of connections that can be simultaneously opened to a given server (default is 2).
proxyName	Name of the proxy server.
proxyPort	Port number for your proxy server.
suppressChannelData	Directive specifying whether the channel will contribute to the ChannelData that is used when creating an ObjRef. Takes a value of true or false (default is false).
useIpAddress	Directive specifying whether the channel shall use IP addresses in the given URLs instead of using the hostname of the server computer. Takes a value of true or false (default is true).
listen	Directive specifying whether activation shall be allowed to hook into the listener service. Takes a value of true or false (default is true).
bindTo	IP address on which the server will listen. Used only on computers with more than one IP address.
machineName	A string that specifies the machine name used with the current channel. This property overrides the useIpAddress property.

In addition, there are some security-related properties that are only supported by the HTTP channel that can be used in conjunction with remoting servers hosted in IIS.

Attribute	Description
useDefaultCredentials	When you use this property on the client side, the security credentials of the currently logged in user are passed to the remoting server hosted in IIS for authentication purposes (actually, it automatically gets the credentials from System.Net.CredentialCache.DefaultCredentials).
useAuthenticated ConnectionSharing	This attribute can be used together with the useDefaultCredentials attribute of the channel on the client side. It enables the server to reuse authenticated connections rather than authenticating each incoming call. This feature is improving performance significantly and works only with default credentials.
unsafeAuthenticated ConnectionSharing	While the useAuthenticatedConnectionSharing attribute works only with the default credentials, setting this attribute to true allows connection sharing for credentials passed by your own to the server (which means not the default credentials). The goal is avoiding authentication with each call, but this time with credentials other than default credentials. This attribute must be used together with the connectionGroupName attribute described next. But pay attention, if you set it to true, unauthenticated clients can possibly authenticate to the server using the credentials of a previously authenticated client. This setting is ignored if the useAuthenticatedConnectionSharing property is set to true.
connectionGroupName	The connection group name specifies the name for the connection pool that is used by the server for authenticated clients when the unsafeAuthenticatedConnectionSharing attribute is set to true. This name/value pair is ignored if unsafeAuthenticatedConnectionSharing is not set to true. If specified, make sure that this name maps to only one authenticated user.

The TCP channel, which is created by specifying <channel ref="tcp">, supports the same properties as the HTTP channel (except the security properties) and the following additional property:

Attribute	Description
rejectRemoteRequests	Indicator specifying whether the server will accept requests from remote systems. When set to true, the server will not accept such requests, only allowing interapplication communication from the local machine.

On the server side, the following entry can be used to specify an HTTP channel listening on port 1234:

```
<channels>
   <channel ref="http" port="1234">
</channels>
```

On the client, you can specify an increased connection limit using the following section:

```
<channels>
   <channel ref="http" port="0" clientConnectionLimit="100">
</channels>
```

ClientProviders/ServerProviders

Underneath each channel property, you can configure nondefault client-side and server-side sink providers and formatter providers.

■ **Caution** When any of these elements are specified, it's important to note that no default providers will be created by the system. This means that appending ?WSDL to the URL will only work if you explicitly specify <provider ref="wsdl" />; otherwise you'll receive an exception stating that "no message has been deserialized."

The .NET Remoting framework is based on messages that travel through various layers. Those layers can be extended or replaced and additional layers can be added. (I discuss layers in more detail in Chapter 11.)

These layers are implemented using so-called message sinks. A message will pass a chain of sinks, each of which will have the possibility to work with the message's content or to even change the message.

Using the ClientProviders and ServerProviders properties in the configuration file, you can specify this chain of sinks through which you want a message to travel and the formatter with which a message will be serialized.

The structure for this property for the server side is as follows:

```
<channels>
  <channel ref="http" port="1234">
     <serverProviders>
        <formatter />
        <provider />
     </serverProviders>
  </channel>
</channels>
```

You may only have one formatter entry but several provider properties. Also note that sequence *does* matter.

The following attributes are common between formatters and providers:

Attribute	Description
ref	Reference for a predefined SinkProvider ("soap", "binary", or "wsdl") or reference to a SinkProvider that has been defined in a configuration file.
type	Attribute that is mandatory when ref has not been specified. Contains the exact type (namespace, class name, assembly) of the SinkProvider's implementation. When the assembly is in the GAC, you have to specify version, culture, and public key information as well.

Here are additional attributes that are optional for formatters:

Attribute	Description
includeVersions	Indicator of whether version information should be included in the requests. Takes a value of true or false (defaults to true for built-in formatters). This attribute changes behavior on the client side.
strictBinding	Indicator of whether the server will look for the exact type (including version) or any type with the given name. Takes a value of true or false (defaults to false for built-in formatters).

In addition to these attributes, both formatters and providers can accept custom attributes, as shown in the following example. You have to check the documentation of your custom sink provider for the names and possible values of such properties:

```
<channels>
  <channel ref="http" port="1234">
     <serverProviders>
        <provider type="MySinks.SampleProvider, Server" myAttribute="myValue" />
            <sampleProp>This is a Sample</sampleProp>
            <sampleProp>This is another Sample</sampleProp>
        </provider>
        <formatter ref="soap" />
     </serverProviders>
  </channel>
</channels>
```

typeFilterLevel Attribute on Formatters

Last but not least, there is one additional attribute I have to mention, the typeFilterLevel attribute, which can be applied on formatters on the server side as well as the client side. Although the attribute already existed in version 1.0 of the .NET Framework, with the trustworthy computing initiative of Microsoft and the principle of *Secure By Default*, the behavior of this attribute from version 1.0 to 1.1 of the .NET Framework has changed.

When objects are sent across the wire using .NET Remoting, they have to be serialized and deserialized. That's true for both MarshalByRef objects, whereby an ObjRef is serialized, and MarshalByValue objects, whereby the whole object is serialized and deserialized.

Theoretically an unauthorized and malicious client could try to exploit the moment of deserialization. Here you can find an outline of threats that might happen with deserialization of objects:

- If a remoting server exposes a method or member that takes a delegate, this delegate may be used to invoke methods on other types that have the same method signature.
- When the remoting server takes a MarshalByRef object from the client, the URL stored in the ObjRef passed from the client to the server can be tampered with so that the server calls back to malicious clients.
- Implementing the ISerializable interface on a serializable type means that code is executed on this type each time it is serialized and deserialized. If a malicious client passes potentially dangerous data on such types, which leads to code execution on the server during deserialization, this can result in security vulnerabilities.
- Often remote servers enable clients to register sponsors. But the number of sponsors that can be registered is not limited, which can lead to denial of service attacks.

These are just a few threats that can be (partially) mitigated using the typeFilterLevel attribute. Basically this attribute has two possible values for type filtering by the runtime: low and full. The effects of each value are described in the following table.

Attribute	Description
Low	With .NET 1.1 this is the default value. The default level permits deserialization of the following types: Remoting infrastructure objects necessary for doing basic remoting tasks Primitive types and reference and value types composed of only primitives Reference- and value-types that are marked as [Serializable] but don't implement ISerializable System-provided types that implement ISerializable but don't make further permission demands System-provided, serializable types in assemblies that are not marked with the APTCA[1] attribute Custom types in strong-named assemblies that are not marked with APTCA Custom types that implement ISerializable without making further demands outside serialization Types that implement ILease but are not MarshalByRef objects ObjRef objects needed for activation of CAO, which can be deserialized on clients but not on servers
Full	The full setting allows deserialization of any remoted types sent across the wire. This means ObjRef objects passed as parameters, objects that implement ISponsor, as well as objects that are inserted between the proxy and client pipeline by the IContributeEnvoySink interface are supported in addition to supported types of the Low deserialization level.

1. APTCA is the acronym for AllowPartiallyTrustedCallersAttribute. This attribute can be applied on strong-named assemblies so that they can be called from partially trusted code, which is not allowed by default.

Concrete things like callbacks to clients through delegates and vice versa will not work by default when deploying .NET Remoting solutions with .NET 1.1. In this case, you have to configure the typeFilterLevel attribute to the *full* setting manually. This means threats mentioned previously are not mitigated any more, and you have to include strong authentication and encryption mechanisms for your .NET Remoting applications to mitigate the threats with other techniques. Security in .NET Remoting solutions is discussed in the next chapter.

You can configure the typeFilterLevel either manually in code or through configuration files. The attribute itself is applied at the formatter level. This means a configuration file with the typeFilterLevel attribute configured might look like the following (the setting must be configured for each channel and each formatter):

```
<configuration>
  <system.runtime.remoting>
    <application>
      <channels>
        <channel ref="tcp" port="1234">
          <serverProviders>
            <formatter ref="binary"
                       typeFilterLevel="Low" />
            <formatter ref="soap"
                       typeFilterLevel="Low" />
          </serverProviders>
        </channel>
      </channels>
      <service>
        <wellknown type="Server.ServerImpl, Server"
                   objectUri="MyServer.rem"
                   mode="Singleton" />
      </service>
    </application>
  </system.runtime.remoting>
</configuration>
```

■ **Note** The preceding configuration file shows the configuration of a remoting server. If you have to configure a client that needs to receive callbacks from the server through delegates, the client's channel receives those callbacks through its own receiving port. It acts as a server for the callbacks. Therefore, if the client wants to be able to receive callbacks, you have to include the <serverProviders> configuration tag for each channel configured in the client to set the typeFilterLevel property.

If you want to configure the typeFilterLevel in code, you have to apply the setting on either the BinaryServerFormatterSinkProvider or the SoapServerFormatterSinkProvider through their generic property collection. The following code snippet demonstrates configuration through code:

```
// configure the formatters for the channel
BinaryServerFormatterSinkProvider formatterBin =
              new BinaryServerFormatterSinkProvider();
formatterBin.TypeFilterLevel = TypeFilterLevel.Full;

SoapServerFormatterSinkProvider formatterSoap =
              new SoapServerFormatterSinkProvider();
formatterSoap.TypeFilterLevel = TypeFilterLevel.Low;

formatterBin.Next = formatterSoap;

// register the channels
IDictionary dict = new Hashtable();
dict.Add("port", "1234");

TcpChannel channel = new TcpChannel(dict, null, formatterBin);
ChannelServices.RegisterChannel(channel);

// register the wellknown service
RemotingConfiguration.RegisterWellKnownServiceType(typeof(ServerImpl),
                              "MyServer.rem", WellKnownObjectMode.Singleton);
```

The preceding code leads to the same configuration as the configuration file with the typeFilterLevel configured shown previously. But never forget that the typeFilterLevel is just one method for mitigating attacks. It doesn't mean that you don't have to employ other security techniques for mitigation of attacks. And very often you have to configure the typeFilterLevel to full because perhaps you need callbacks to the client or similar things. In this case, you have to include additional security gatekeepers to protect against threats like strong authentication and encryption of traffic. You can find more about security and .NET Remoting in Chapter 5.

Versioning Behavior

Depending on the setting of the includeVersion attribute on the client-side formatter and the strictBinding attribute on the server-side formatter, different methods for creating instances of the given types are employed:

includeVersions	strictBinding	Resulting Behavior
true	true	The exact type is loaded, or a TypeLoadException is thrown.
false	true	The type is loaded using only the type name and the assembly name. A TypeLoadException is thrown if this type doesn't exist.
true	false	The exact type is loaded if present; if not, the type is loaded using only the type name and the assembly name. If the type doesn't exist, a TypeLoadException is thrown.[2]
false	false	The type is loaded using only the type name and the assembly name. A TypeLoadException is thrown if this type doesn't exist.

2. The strictBinding attribute works only if the assemblies with the seralizable types are installed in the Global Assembly Cache. If they are not installed in the GAC, you get a TypeLoadException even if you configure the includeVersions and strictBinding settings like that.

Binary Encoding via HTTP

As you already know, the default HTTP channel will use a SoapFormatter to encode the messages. Using configuration files and the previously mentioned properties, you can easily switch to a BinaryFormatter for an HTTP channel.

On the server side, you use the following section in your configuration file:

```
<channels>
  <channel ref="http" port="1234">
     <serverProviders>
        <formatter ref="binary" />
     </serverProviders>
  </channel>
</channels>
```

And on the client side, you can take the following configuration file snippet:

```
<channels>
  <channel ref="http">
     <clientProviders>
        <formatter ref="binary" />
     </clientProviders>
  </channel>
</channels>
```

■ **Note** The server-side entry is not strictly necessary, because the server-side HTTP channel automatically uses both formatters and detects which encoding has been chosen at the client side.

Service

The <service> property in the configuration file allows you to register SAOs and CAOs that will be made accessible by your server application. This section may contain a number of <wellknown> and <activated> properties.

The main structure of these entries is as follows:

```
<configuration>
   <system.runtime.remoting>
      <application>
         <service>
            <wellknown />
            <activated />
         </service>
      </application>
   </system.runtime.remoting>
</configuration>
```

Wellknown

Using the <wellknown> property in the server-side configuration file, you can specify SingleCall and Singleton objects that will be provided by your server. This property supports the same attributes that can also be specified when calling `RemotingConfiguration.RegisterWellKnownServiceType()`, as listed here:

Attribute	Description
type	The type information of the published class in the form "<namespace>.<classname>, <assembly>". When the assembly is in the GAC, you have to specify version, culture, and public key information as well.
mode	Indicator specifying object type. Can take "Singleton" or "SingleCall".
objectUri	The endpoint URI for calls to this object. When the object is hosted in IIS (shown later in this chapter), the URI has to end with .soap or .rem to be processed correctly, as those extensions are mapped to the .NET Remoting framework in the IIS metabase.
displayName	Optional attribute that specifies the name that will be used inside the .NET Framework Configuration Tool.

Using the following configuration file, the server will allow access to a CustomerManager object via the URI `http://<host>:1234/CustomerManager.soap`.

```
<configuration>
  <system.runtime.remoting>
    <application>
      <channels>
        <channel ref="http" port="1234" />
      </channels>
      <service>
        <wellknown mode="Singleton"
                   type="Server.CustomerManager, Server"
                   objectUri="CustomerManager.soap" />
      </service>

    </application>
  </system.runtime.remoting>
</configuration>
```

Activated

The <activated> property allows you to specify CAOs in the server-side configuration file. As the full URI to this object is determined by the application name, the only attribute that has to be specified is the type to be published.

Attribute	Description
type	The type information of the published class in the form "<namespace>.<classname>, <assembly>". When the assembly is in the GAC, you have to specify version, culture, and public key information as well.

The following example allows a client to create an instance of MyClass at `http://<hostname>:1234/`:

```
<configuration>
   <system.runtime.remoting>
      <application>
         <channels>
            <channel ref="http" port="1234" />
         </channels>
         <service>
            <activated type="MyObject, MyAssembly"/>
         </service>
      </application>
   </system.runtime.remoting>
</configuration>
```

Client

The client-side counterpart to the <service> property is the <client> configuration entry. Its primary structure is designed to look quite similar to the <service> entry:

```
<configuration>
   <system.runtime.remoting>
      <application>
         <client>
            <wellknown />
            <activated />
         </client>
      </application>
   </system.runtime.remoting>
</configuration>
```

When using CAOs, the <client> property has to specify the URI to the server for all underlying <activated> entries.

■ **Note** When using CAOs from more than one server, you have to create several <client> properties in your configuration file.

Attribute	Description
url	The URL to the server, mandatory when using CAOs
displayName	Attribute that is used in the .NET Framework Configuration Tool

Wellknown

The <wellknown> property is used to register SAOs on the client and allows you to use the `new` operator to instantiate references to remote objects. The client-side <wellknown> entry has the same attributes as the call to `Activator.GetObject()`, as listed here:

Attribute	Description
url	The full URL to the server's registered object.
type	Type information in the form "<namespace>.<classname>, <assembly>". When the target assembly is registered in the GAC, you have to specify version, culture, and public key information as well.
displayName	Optional attribute that is used in the .NET Framework Configuration Tool.

When registering a type to be remote, the behavior of the `new` operator will be changed. The framework will intercept each call to this operator and check whether it's for a registered remote object. If this is the case, a reference to the server will be created instead of an instance of the local type.

When the following configuration file is in place, you can simply write `CustomerManager x = new CustomerManager()` to obtain a remote reference.

```
<configuration>
  <system.runtime.remoting>
    <application>

      <client>
        <wellknown type="Server.CustomerManager, Client"
                   url="http://localhost:1234/CustomerManager.soap" />
      </client>

    </application>
  </system.runtime.remoting>
</configuration>
```

Activated

This is the client-side counterpart to the <activated> property on the server. As the URL to the server has already been specified in the <client> entry, the only attribute to specify is the type of the remote object.

Attribute	Description
type	The type information in the form "<namespace>.<classname>, <assembly>". When the target assembly is registered in the GAC, you have to specify version, culture, and public key information as well.

Data from this entry will also be used to intercept the call to the `new` operator. With a configuration file like the following, you can just write `MyRemote x = new MyRemote()` to instantiate a server-side CAO.

```
<configuration>
  <system.runtime.remoting>
    <application>

      <client url="http://localhost:1234/MyServer">
        <activated type="Server.MyRemote, Client" />
      </client>

    </application>
  </system.runtime.remoting>
</configuration>
```

What About Interfaces?

In the previous chapters, and earlier in this chapter, I stated that using SoapSuds.exe is not the best choice for your real-world applications for a number of reasons. However, if you want to use configuration files in the way I've described them earlier, then you will unfortunately have to resort to shipping your complete implementation assembly or to using SoapSuds.exe. Neither of these solutions is really favorable for most applications.

There is, however, a middle-ground solution for this problem: you can define interfaces in a shared DLL and use a little helper class to acquire remote references to these interfaces without hard coding any information. It's still not as transparent as if you were using just an overloaded `new` operator, but it's as close as you can get.

It will allow you for example to use code like the following together with a matching configuration file to create a remote reference (proxy) to an IRemoteCustomerManager object:

```
IRemoteCustomerManager mgr = (IRemoteCustomerManager)
  RemotingHelper.CreateProxy(typeof(IRemoteCustomerManager));
```

You can see the code for this helper class in Listing 4-5. Its method, `InitTypeCache()`, iterates over the configured interfaces and adds each found interface to an internal dictionary object. As soon as `CreateProxy()` is called, it checks this dictionary for the specified type to create a remote reference based on the configured URL.

Listing 4-5. *The RemotingHelper*

```
using System;
using System.Collections;
using System.Runtime.Remoting;

class RemotingHelper
{
   private static IDictionary _wellKnownTypes;
```

```
   public static Object CreateProxy(Type type)
   {
      if (_wellKnownTypes==null) InitTypeCache();
      WellKnownClientTypeEntry entr =
             (WellKnownClientTypeEntry) _wellKnownTypes[type];
      if (entr == null)
      {
         throw new RemotingException("Type not found!");
      }
      return Activator.GetObject(entr.ObjectType,entr.ObjectUrl);
   }

   public static void InitTypeCache()
   {
      Hashtable types= new Hashtable();
      foreach (WellKnownClientTypeEntry entr in
         RemotingConfiguration.GetRegisteredWellKnownClientTypes())
      {
         if (entr.ObjectType == null)
         {
            throw new RemotingException("A configured type could not " +
               "be found. Please check spelling in your configuration file.");
         }
         types.Add (entr.ObjectType,entr);
      }
      _wellKnownTypes = types;
   }
}
```

The following is a sample configuration file that can be used with the RemotingHelper to configure the interface IRemoteCustomerManager to use a remote implementation:

```
<configuration>
  <system.runtime.remoting>
    <application>
      <client>
        <wellknown type="General.IRemoteCustomerManager, General"
                   url="http://localhost:1234/CustomerManager.rem" />
      </client>
    </application>
  </system.runtime.remoting>
</configuration>
```

You have to take care when using configuration files like these because the .NET Framework by default doesn't support the creation of remote references for configured interfaces. This is *only* possible when using the RemotingHelper class, and someone who reads your code for the first time might not know about this.

■ **Note** Most of the remaining samples in this book will use this RemotingHelper class to create remote references.

Using the IPC Channel in .NET Remoting 2.0

With the next version, the .NET Framework 2.0 and Visual Studio 2005, Microsoft adds two very important features to the .NET Remoting infrastructure. First of all, security is an integral part of .NET Remoting 2.0 as you will see in the next chapter, and secondly, a new channel is included. In this chapter, I want to focus on the new channel and the configuration options coming with it.

The new channel, the so-called IPC channel, is a channel optimized for interprocess communication between two processes running on the same machine. That's something that is missing in .NET Remoting 1.x—without custom implementations, you have to use either the TCP or the HTTP channel for interprocess communication. Actually, opening a TCP or a HTTP port and communicating through the "networking infrastructure," although both applications run on the same machine, constitute a really unnecessary overhead. Regardless of performance reasons, furthermore for security reasons you have to make sure that the port is blocked so that communication is really limited to the local machine.

In .NET Remoting 2.0, you can use the IpcChannel for implementing interprocess communication between processes running on the same machine. With this you can get a performance boost for your applications as well as better security because the IpcChannel works only within machine boundaries. In the following sample, I will show you how to use the IPC channel with .NET 2.0.

■ **Note** For .NET Remoting 1.x, a so-called Named Pipe channel sample, which can be used for optimized interprocess communication between processes running on the same machine, is available for download at `http://www.gotdotnet.com/Community/UserSamples/Details.aspx?SampleGuid=43a1ef11-c57c-45c7-a67f-ed68978f3d6d`. This sample has been developed by a Microsoft employee with close ties to the .NET Remoting team, but is not officially supported.

In my example, I will configure the IpcChannel programmatically on the client side and using configuration files on the server side so that you can see both ways of working with the channel. Let's start with the shared assembly for the client and the server, which can be seen in Listing 4-6.

Listing 4-6. *The Shared Assembly Defining Interfaces*

```
using System;
using System.Collections.Generic;
using System.Text;

namespace RemotedType
{
```

```
    public interface IRemotedType
    {
        void DoCall(string message, int counter);
    }
}
```

The server is going to implement this fairly simple interface and will do nothing other than taking the message passed through the method parameters and outputting it the number of times specified in the counter parameter (see Listing 4-7).

Listing 4-7. *The Server Implementation*

```
using System;
using System.Collections.Generic;
using System.Text;
using System.Runtime.Remoting;
using System.Runtime.Remoting.Channels;
using System.Runtime.Remoting.Channels.Ipc;
using System.Runtime.Remoting.Messaging;
using System.Runtime.Remoting.Contexts;
using System.Runtime.Remoting.Activation;

using System.Security;
using System.Security.Principal;

namespace RemotingServer
{
  public class MyRemoteObject : MarshalByRefObject, RemotedType.IRemotedType
  {
    public void DoCall(string message, int counter)
    {
      // get some information about the caller's context
      IIdentity remoteIdentity =
          CallContext.GetData("__remotePrincipal") as IIdentity;
      if (remoteIdentity != null)
      {
        System.Console.WriteLine("Authenticated user:\n-){0}\n-){1}",
                                 remoteIdentity.Name,
                                 remoteIdentity.AuthenticationType.ToString());
      }
      else
      {
        System.Console.Write("!! Attention, not authenticated !!");
      }

      // just do the stupid work
      for (int i = 0; i < counter; i++)
      {
```

```
      System.Console.WriteLine("You told me to say {0}: {1}!",
                                counter.ToString(), message);
    }
  }
}

public class RemoteServerApp
{
  static void Main(string[] args)
  {
    try
    {
      System.Console.WriteLine("Configuring server...");
      System.Runtime.Remoting.RemotingConfiguration.Configure(
                                 "RemotingServer.exe.config");

      System.Console.WriteLine("Configured channels:");
      foreach (IChannel channel in ChannelServices.RegisteredChannels)
      {
        System.Console.WriteLine("Registered channel: " + channel.ChannelName);
        if (channel is IpcChannel)
        {
          if (((IpcChannel)channel).ChannelData != null)
          {
            ChannelDataStore dataStore =
               (ChannelDataStore)((IpcChannel)channel).ChannelData;
            foreach (string uri in dataStore.ChannelUris)
            {
              System.Console.WriteLine("-) Found URI: " + uri);
            }
          }
          else
          {
            System.Console.WriteLine("-) No channel data");
          }
        }
        else
        {
          System.Console.WriteLine("-) not a IpcChannel data store");
        }
      }

      System.Console.WriteLine("--- waiting for requests...");
      System.Console.ReadLine();
      System.Console.WriteLine("Finished!!");
    }
    catch (Exception ex)
    {
```

```
        System.Console.WriteLine("Error while configuring server: " + ex.Message);
        System.Console.ReadLine();
      }
    }
  }
}
```

The first class is the server implementation itself. More interesting are the parts in the main method of the application where the configuration is loaded. After the configuration file has been configured through `RemotingConfiguration.Configure()`, the server determines the configured channels and outputs the details (that's something that works with .NET 1.x, too).

All registered channels are iterated, and for each registered IpcChannel details like the name or its URI will be printed to the screen. When starting the server, the output looks like the one in Figure 4-9.

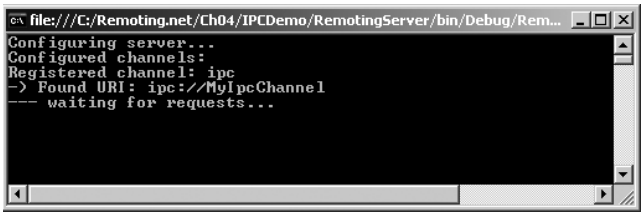

Figure 4-9. *The console output window of Visual Studio 2005*

■ **Note** When working with Visual Studio 2005, the first console application launched for debugging in your solution is started in the console window. Any further console applications launched afterwards will be started in their own console window as usual.

In the following code snippet, you can see the server's configuration. The configuration is not much different from usual .NET Remoting configurations.

```
<?xml version="1.0" encoding="utf-8" ?>
<configuration>
  <system.runtime.remoting>
    <application name="MyServer">
      <service>
        <wellknown type="RemotingServer.MyRemoteObject, RemotingServer"
                   objectUri="MyObject.rem"
                   mode="SingleCall" />
      </service>
      <channels>
        <channel ref="ipc" portName="MyIpcChannel" />
      </channels>
```

```
    </application>
  </system.runtime.remoting>
</configuration>
```

There is just one thing to note: instead of configuring a port number for an IPC channel, it gets a unique name. This name identifies the IPC port (that said, each named pipe port needs its own unique name) and is used by the client to connect to the server as you will see in the client's implementation in Listing 4-8.[3] Therefore, the name must be unique at the machine level to avoid conflicts between your application and other applications running on the same machine.

In the client application, the IPC channel is manually configured in the source code. Listing 4-8 shows the client's implementation.

Listing 4-8. *The .NET 2.0 Client Implementation*

```
using System;
using System.Collections.Generic;
using System.Runtime.Remoting;
using System.Runtime.Remoting.Channels;
using System.Runtime.Remoting.Channels.Ipc;
using System.Runtime.Remoting.Activation;
using System.Text;

using RemotedType;

namespace RemotingClient
{
  class Program
  {
    static void Main(string[] args)
    {
      try
      {
        System.Console.WriteLine("Configuring channel...");
        IpcClientChannel clientChannel = new IpcClientChannel();
        ChannelServices.RegisterChannel(clientChannel);

        System.Console.WriteLine("Configuring remote object...");
        IRemotedType TheObject = (IRemotedType)Activator.GetObject(
            typeof(RemotedType.IRemotedType),
            "ipc://MyIpcChannel/MyObject.rem");

        System.Console.WriteLine("Please enter data, 'exit' quits the program!");
        string input = string.Empty;
```

3. If you end the pipe's name with "$", other users have to know the exact name and cannot search for it.

```
        do
        {
          System.Console.Write(">> Enter text: ");
          input = System.Console.ReadLine();
          if (string.Compare(input, "exit", true) != 0)
          {
            System.Console.Write(">> Enter number: ");
            int c = Int32.Parse(System.Console.ReadLine());
            TheObject.DoCall(input, 2);
          }
        } while (string.Compare(input, "exit", true) != 0);
      }
      catch (Exception ex)
      {
        System.Console.WriteLine("Exception: " + ex.Message);
        System.Console.ReadLine();
      }
    }
  }
}
```

First of all, the client creates a client channel and registers it through the ChannelServices. Afterwards it connects to the server using `Activator.GetObject()`. Instead of using the tcp or http protocol prefix in the URL, it uses ipc. You don't need to specify a machine name in the URL because communication happens on the local machine. The first part of the URL is the name of the IPC port as it has been specified in the server's configuration. The second part is the object URI as usual for remoting objects.

Basically that's it. The most important difference between the IPC channel and the other channels is that you have to specify a unique name for the channel as well as the type of URL used for connecting to the channel. In Figures 4-10 and 4-11 you can see the client and server in action.

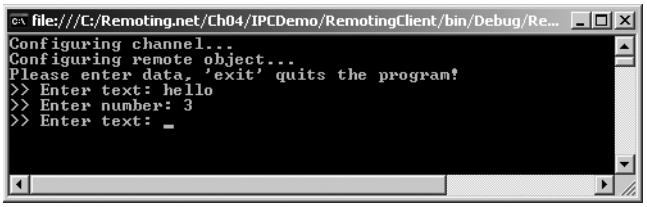

Figure 4-10. *The client application in action*

file:///C:/Remoting.net/Ch04/IPCDemo/RemotingServer/bin/Debug/Remoting...
Configuring server...
Configured channels:
Registered channel: ipc
-> Found URI: ipc://MyIpcChannel
--- waiting for requests...
!! Attention, not authenticated !!You told me to say 2: hello!
You told me to say 2: hello!

Figure 4-11. *The server application in action*

Deployment

In contrast to some other frameworks (Java RMI, J2EE EJB, COM+, and so on), .NET Remoting allows you to choose quite freely how you want to deploy your server application. You can publish the objects in any kind of managed application—console, Windows Forms, and Windows services—or host them in IIS.

Console Applications

Deploying servers as .NET console applications is the easiest way to get started; every example up to this point has been designed to run from the console. The features are easily observable: instant debug output and starting, stopping, and debugging is possible using the IDE.

Production applications nevertheless have different demands: when using console applications, you need to start the program after logging on to a Windows session. Other possible requirements such as logging, authentication, and encryption are hard to implement using this kind of host.

Windows Services

If you don't want to host the objects in IIS, classic Windows services are the way to go. Visual Studio .NET, and the .NET Framework in general, make it easy for you to develop a Windows service application. They take care of most issues, starting from the installation of the service to encapsulating the communication between the service control manager and your service application.

Integrating remoting in Windows services can also be viewed from another standpoint: when your primary concern is to write a Windows service—for example, to provide access to privileged resources—you can easily implement the communication with your clients using .NET Remoting. This is somewhat different from conventional approaches in the days before .NET, which forced you to define distinct communication channels using named pipes, sockets, or the COM ROT.

Cross-process remoting on a local machine using a TCP channel ought to be fast enough for most applications.

Porting to Windows Services

In the .NET Framework, a Windows service simply is a class that extends System.ServiceProcess.ServiceBase. You basically only have to override `OnStart()` to do something useful.

A baseline Windows service is shown in Listing 4-9.

Listing 4-9. *A Baseline Windows Service*

```
using System;
using System.Diagnostics;
using System.ServiceProcess;

namespace WindowsService
{
   public class DummyService : System.ServiceProcess.ServiceBase
   {
      public static String SVC_NAME = "Some dummy service";

      public DummyService ()
      {
         this.ServiceName = SVC_NAME;
      }

      static void Main()
      {
           // start the service
           ServiceBase.Run(new DummyService());
      }

      protected override void OnStart(string[] args)
      {
           // do something meaningful
      }

      protected override void OnStop()
      {
           // stop doing anything meaningful ;)
      }
   }
}
```

A service like this will not be automatically installed in the service manager, so you have to provide a special Installer class that will be run during the execution of installutil.exe (from the .NET command prompt).

Listing 4-10 shows a basic service installer that registers the service to be run using the System account and started automatically during boot-up.

Listing 4-10. *A Basic Windows Service Installer*

```
using System;
using System.Collections;
using System.Configuration.Install;
```

```
using System.ServiceProcess;
using System.ComponentModel;
using WindowsService;

[RunInstallerAttribute(true)]
public class MyProjectInstaller: Installer
{
   private ServiceInstaller serviceInstaller;
   private ServiceProcessInstaller processInstaller;

   public MyProjectInstaller()
   {
      processInstaller = new ServiceProcessInstaller();
      serviceInstaller = new ServiceInstaller();

      processInstaller.Account = ServiceAccount.LocalSystem;
      serviceInstaller.StartType = ServiceStartMode.Automatic;
      serviceInstaller.ServiceName = DummyService.SVC_NAME;

      Installers.Add(serviceInstaller);
      Installers.Add(processInstaller);
   }
}
```

The installer has to be in your main assembly and has to have [RunInstallerAttribute(true)] set. After compiling the preceding C# files, you will have created a baseline Windows service that can be installed with installutil.exe.

When porting the remoting server to become a Windows service, you might want to extend the base service to also allow it to write to the Windows event log. Therefore, you have to add a static variable of type EventLog to hold an instance acquired during `void Main()`. As an alternative, you could also set the `AutoLog` property of the service and use the static method `EventLog.WriteEntry()`. You will also have to extend `onStart()` to configure remoting to allow the handling of requests as specified in the configuration file. The complete source code for the Windows service–based remoting server is shown in Listing 4-11.

Listing 4-11. *A Simple Windows Service to Host Your Remote Components*

```
using System;
using System.Diagnostics;
using System.ServiceProcess;
using System.Runtime.Remoting;

namespace WindowsService
{
   public class RemotingService : System.ServiceProcess.ServiceBase
   {
      private static EventLog evt = new EventLog("Application");
      public static String SVC_NAME = ".NET Remoting Sample Service";
```

```
        public RemotingService()
        {
            this.ServiceName = SVC_NAME;
        }

        static void Main()
        {
            evt.Source = SVC_NAME;
            evt.WriteEntry("Remoting Service intializing");
            ServiceBase.Run(new RemotingService());
        }

        protected override void OnStart(string[] args)
        {
            evt.WriteEntry("Remoting Service started");
            String filename =
                   AppDomain.CurrentDomain.SetupInformation.ConfigurationFile;
            RemotingConfiguration.Configure(filename);
        }

        protected override void OnStop()
        {
            evt.WriteEntry("Remoting Service stopped");
        }
    }
}
```

In two separate classes, you'll then provide the implementation of the MarshalByRefObject CustomerManager and an installer, following the preceding sample.

When this program is run in the IDE, you'll see the biggest disadvantage to developing Windows services, the message box that will pop up, telling you that you won't get automatic debugging support from Visual Studio .NET IDE (see Figure 4-12).

Windows Service Start Failure
Cannot start service from the command line or a debugger. A Windows Service must first be installed (using installutil.exe) and then started with the ServerExplorer, Windows Services Administrative tool or the NET START command.
OK

Figure 4-12. *Trying to start a Windows service from the IDE*

To install this application as a Windows service so that it can be started and stopped via the Services MMC snap-in or Server Explorer, you have to run installutil.exe, which is best done from a .NET command prompt. Running this application without the option /LogToConsole=false will produce a lot of information in case something goes wrong. You can see the command line for installing the service in Figure 4-13.

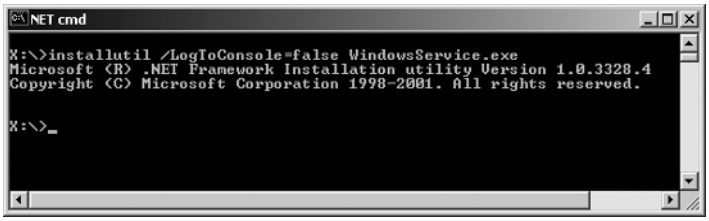
NET cmd
X:\>installutil /LogToConsole=false WindowsService.exe
Microsoft (R) .NET Framework Installation utility Version 1.0.3328.4
Copyright (C) Microsoft Corporation 1998-2001. All rights reserved.
X:\>

Figure 4-13. *Installing a service using installutil.exe*

After successfully installing the service, you will be able to start it using the Microsoft Management Console, as shown in Figure 4-14.

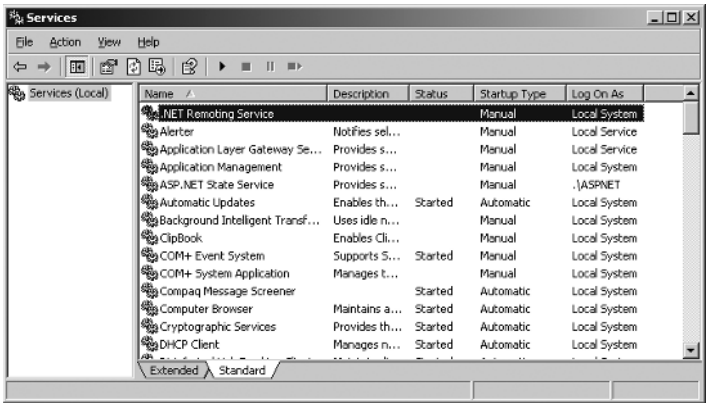
Services
File Action View Help
Services (Local)
Name Description Status Startup Type Log On As
.NET Remoting Service Manual Local System
Alerter Notifies sel... Manual Local Service
Application Layer Gateway Se... Provides s... Manual Local Service
Application Management Provides s... Manual Local System
ASP.NET State Service Provides s... Manual .\ASPNET
Automatic Updates Enables th... Started Automatic Local System
Background Intelligent Transf... Uses idle n... Manual Local System
ClipBook Enables Cli... Manual Local System
COM+ Event System Supports S... Started Manual Local System
COM+ System Application Manages t... Manual Local System
Compaq Message Screener Started Automatic Local System
Computer Browser Maintains a... Started Automatic Local System
Cryptographic Services Provides th... Started Automatic Local System
DHCP Client Manages n... Started Automatic Local System
Extended Standard

Figure 4-14. *The service has been installed successfully.*

After starting the service, you can see the output in the EventLog viewer, which is shown in Figure 4-15.

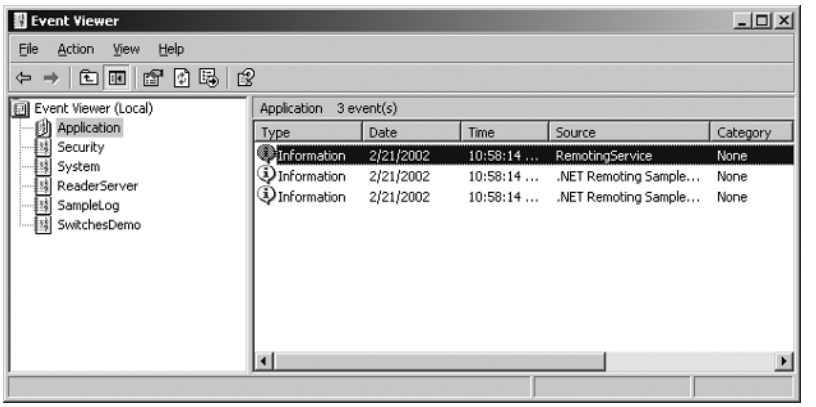

Figure 4-15. *The server's output in the EventLog*

You can now use the same client as in the previous example to connect to the server. After stopping it in the MMC, you can uninstall the service with the installutil.exe /U option, as shown in Figure 4-16.

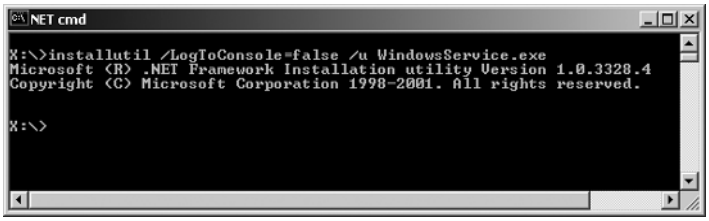
NET cmd
X:\>installutil /LogToConsole=false /u WindowsService.exe
Microsoft (R) .NET Framework Installation utility Version 1.0.3328.4
Copyright (C) Microsoft Corporation 1998-2001. All rights reserved.
X:\>

Figure 4-16. *Uninstalling a service using installutil.exe*

Debugging a Windows Service

As you've seen previously, Windows services cannot be automatically started in debug mode from within VS .NET. Instead, you'd have to resort to one of two possible workarounds to debug your service. Your first option is to change your server application's `void Main()` to accept additional parameters that specify whether or not the application should run as a service at all. This is shown in Listing 4-12.

Listing 4-12. *This Main() Allows Switching Between Service/Nonservice.*

```
static void Main(string[] args)
{
   evt.Source = SVC_NAME;
   evt.WriteEntry("Remoting Service intializing");
   if (args.Length>0 && args[0].ToUpper() == "/NOSERVICE")
   {
      RemotingService svc = new RemotingService();
      svc.OnStart(args);
      Console.WriteLine("Service simulated. Press <enter> to exit.");
      Console.ReadLine();
      svc.OnStop();
   }
   else
   {
      System.ServiceProcess.ServiceBase.Run(new RemotingService());
   }
}
```

You can then right-click your project in Visual Studio .NET to set its properties. To pass a command-line parameter to your application in debug mode, you have to go to Configuration Properties ➤ Debugging ➤ Command Line Arguments and enter the value you expect, which is /NOSERVICE in the preceding sample. This is illustrated in Figure 4-17.

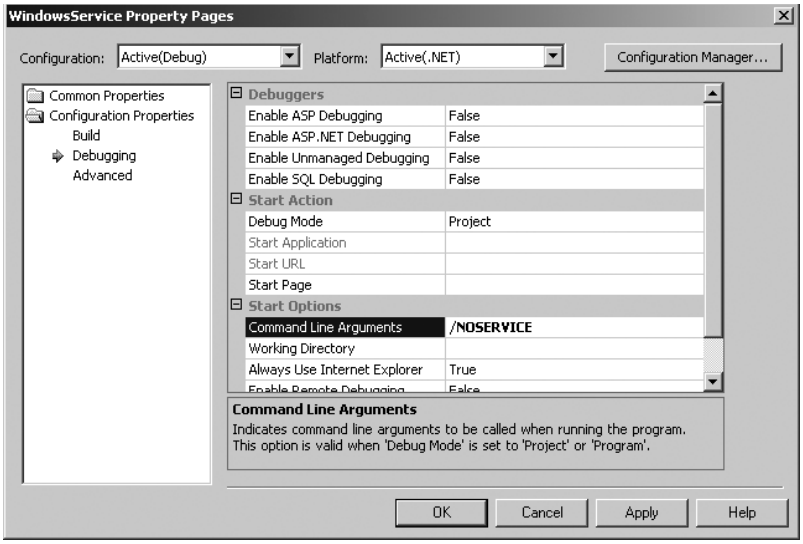

Figure 4-17. *Configuring the command-line parameters*

You can also change your application's type in Common Properties to run as a console application so that you can see its output. After these changes, you can simply hit F5 to debug your program.

Note, however, that this program will behave differently from a real Windows service. It will, for example, run with the currently logged-on user's credentials. If you use a highly privileged account for your work, such as a member of the Administrators group, then your service will also inherit these access rights during debugging. As soon as you deploy your application as a real Windows service, it might run with a limited set of security privileges and might therefore react differently.

If you want to debug a service in its real runtime state, you have to use a slightly more complex approach. First, you have to install your service using installutil.exe, and start it via the Services MMC. After the service is running, you can switch to your project in Visual Studio .NET, and select Debug ➤ Processes. This allows you to attach the VS .NET debugger to a running process, including Windows services.

In the Processes dialog box, which is shown in Figure 4-18, you first have to select the checkbox option Show system processes. After this, you can select your newly created Windows service from the list of running processes and click the Attach button.

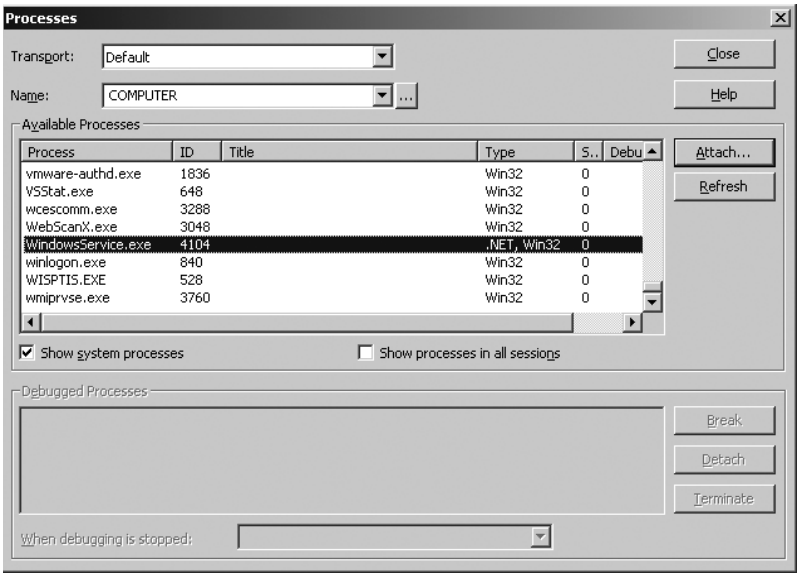

Figure 4-18. *Attaching to a running process*

When attaching to a running process, you have to inform Visual Studio .NET about the kind of debugging you'd like to perform. For debugging Windows services, you'll make sure that at least the Common Language Runtime option is checked as shown in Figure 4-19.

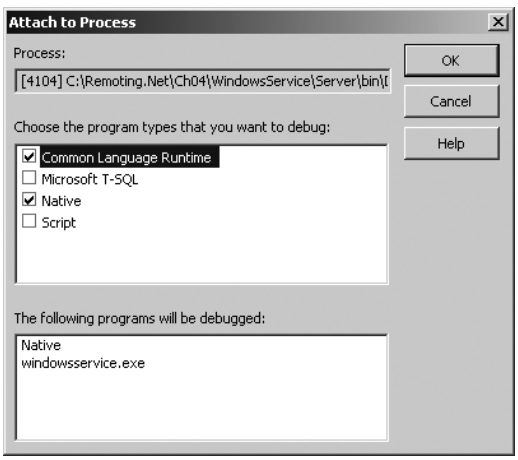

Figure 4-19. *Selecting the type of program for debugging*

After this selection, the Visual Studio .NET debugger will be attached to your Windows service process. This means that you can now set breakpoints, view variable contents, and so

on, just as if you'd started the debugging session using F5. You have to mind, however, that a clean detachment of the debugger is not possible: as soon as you stop the debugger, the debuggee will be forcefully terminated.

Deployment Using IIS

Choosing IIS to host .NET Remoting components allows you to focus closely on application design without wasting any time with developing your own servers, authentication, and security channels, because IIS will take care of these features for you. In addition, IIS will support an optimized threading and HTTP processing model.

■ **Tip** Use Internet Information Server to host your remote components for best scalability and ease of maintenance.

IIS can be configured to take care of a lot of aspects of real-world remoting. Authentication, including the use of Thread.CurrentPrincipal to employ role-based security checks, can be provided with standard Web-based methods. Depending on your architecture and security needs, either basic authentication or NT challenge/response can be chosen. Encryption can be implemented in a secure and simple way using SSL certificates, which need to be installed in IIS.

Hosting in IIS forces you to use the HTTP channel, which—even though you can use the binary formatter instead of the SOAP formatter—will be slower than the TCP channel. In practice however, the performance difference is negligible for most business applications and the increased scalability definitely offsets this initial drawback.

Of course, your needs will dictate the solution to choose. If you expect several thousand calls per second, be sure to check whether any version will meet your demands on the given hardware. If you're going to use the same server-side classes in your in-house LAN application and for outside access via HTTP, you can still set up a different server application using TCP channel for local use and handle the public access via IIS.

Nothing beats hosting of remote objects in IIS in terms of ease of use. You just implement a MarshalByRefObject in a class library project, create a virtual directory in IISAdmin (MMC), create a virtual subdirectory called bin, and put your DLL there. You then have to create a configuration file called web.config and put this in your virtual directory. You don't even need to restart IIS when deploying your remoting components. When you want to update your server-side assembly, you can just overwrite it in the bin subdirectory, as neither the DLL nor the configuration file is locked by IIS.

When the configuration is changed by updating web.config, IIS will automatically reread this information and adopt its behavior. Moving from a staging or testing server to your production machine only involves copying this subdirectory from one host to the other. This is what Microsoft means when talking about xcopy deployment!

Designing and Developing Server-Side Objects for IIS

For the following examples, I also use the RemotingHelper as shown before. The server-side implementation is a little bit easier than the former ways of deployment because you only have to implement the interface in a MarshalByRefObject without any startup code for the server.

Listing 4-13 shows everything you need to code when using IIS to host your components.

Listing 4-13. *Server-Side Implementation of the SAO*

```
using System;
using General;

namespace Server
{
   class CustomerManager: MarshalByRefObject, IRemoteCustomerManager
   {
      public Customer GetCustomer(int id)
      {
         Customer tmp = new Customer();
         tmp.FirstName = "John";
         tmp.LastName = "Doe";
         tmp.DateOfBirth = new DateTime(1970,7,4);
         return tmp;
      }
   }
}
```

After compiling this assembly to a DLL, you can proceed with setting up Internet Information Server.

Preparing Internet Information Server

Before being able to use IIS as a container for server-side objects, you have to configure several aspects of IIS. At the very least you have to create a new virtual root and set the corresponding permissions and security requirements.

Creating a Virtual Root

An IIS virtual root will allow a certain directory to be accessed using a URL. You can basically specify that the URL `http://yourhost/yourdirectory` will be served from c:\somedirectory, whereas the standard base URL `http://yourhost/` is taken from c:\inetpub\wwwroot by default.

You create virtual roots using the MMC, which you bring up by selecting Start ➤ Programs ➤ Administrative Tools ➤ Internet Services Manager. In the Internet Services Manager you will see Default Web Site if you're running workstation software (when using server software, you'll probably see more Web sites).

Right-click Default Web Site and choose New ➤ Virtual Directory. The Virtual Directory Creation Wizard appears. Use this wizard to specify an alias and the directory it will point to. For usage as a remoting container, the default security permissions (Read and Run scripts) are sufficient.

In the directory to which the new IIS virtual directory points, you have to place your .NET Remoting configuration file. As the server will automatically watch for this file and read it, you have to call it web.config—you cannot specify a custom name.

Below this directory you can now create a folder called bin\ and place your assemblies there. As an alternative, you can place your assemblies in the GAC using gacutil.exe, but remember

that you have to specify the SAO assemblies' strong names in the configuration file in this case. This is normally not recommended if your server-side DLL is just used by a single application, but is needed to provide versioning services for CAOs, as you'll see in Chapter 8.

Deployment for Anonymous Use

Before being able to convert the former example to IIS-based hosting, you have to create a new directory and an IIS virtual directory called MyServer (this will be used in the URL to the remote objects) according to the description previously given.

In the Internet Services Manager MMC, access the properties by right-clicking the newly created virtual directory, choosing the Directory Security tab, and clicking Edit. The window shown in Figure 4-20 will open, enabling you to set the allowed authentication methods. Make sure that Allow Anonymous Access is checked.

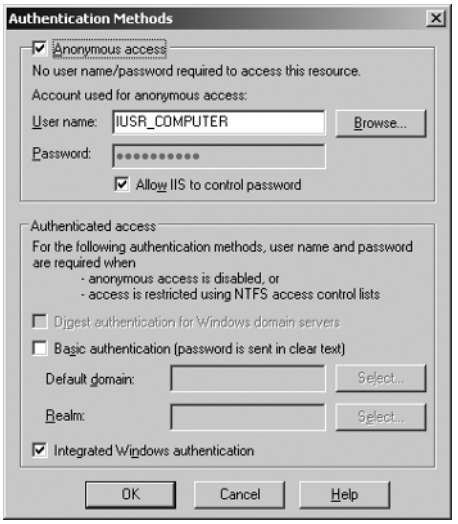

Figure 4-20. *Configuring authentication methods*

In the configuration file, it's important that you don't specify an application name, as this property will be automatically determined by the name of the virtual directory.

■ **Note** When specifying channel information in a configuration file that will be used in IIS—for example, to provide a different sink chain—be sure not to include port numbers, as they will interfere with IIS's internal connection handling. When the server's load reaches a certain point, IIS may start more than one instance for handling a remote object. If you have bound to a secondary port in your configuration file, this port will already be locked by the previous instance.

```
<configuration>
  <system.runtime.remoting>
    <application>
      <service>
        <wellknown mode="Singleton"
                   type="Server.CustomerManager, Server"
                   objectUri="CustomerManager.rem" />

      </service>
    </application>
  </system.runtime.remoting>
</configuration>
```

After putting the configuration file in your virtual directory, you have to copy the assemblies (both the shared one and the server's implementation) into its bin subdirectory. The resulting structure should be like this:

Directory	Contents
x:\<path_to_your_virtual_dir>	web.config
x:\<path_to_your_virtual_dir>\bin	Assemblies that contain your remote objects and assemblies upon which your objects depend if they are not in the GAC

The client is basically the same as in the previous examples. I use a shared interface approach and reference the assembly in which it is contained. The client itself is quite simple, as shown in Listing 4-14.

Listing 4-14. *An Anonymous Client*

```
using System;
using System.Runtime.Remoting;
using General;  // from General.DLL
using Server; // from server.cs

namespace Client
{
   class Client
   {
      static void Main(string[] args)
      {
         String filename =
             AppDomain.CurrentDomain.SetupInformation.ConfigurationFile;
         RemotingConfiguration.Configure(filename);

         IRemoteCustomerManager mgr = (IRemoteCustomerManager)
             RemotingHelper.CreateProxy(typeof(IRemoteCustomerManager));
         Console.WriteLine("Client.Main(): Reference to CustomerManager " +
             "acquired");
```

```
            Customer cust = mgr.GetCustomer(4711);
            int age = cust.GetAge();
            Console.WriteLine("Client.Main(): Customer {0} {1} is {2} years old.",
               cust.FirstName,
               cust.LastName,
               age);

            Console.ReadLine();
         }
      }
}
```

In contrast to the examples earlier in this chapter, the client's configuration file needs to be changed to contain the correct URL to the newly deployed components.

```
<configuration>
  <system.runtime.remoting>
    <application>
      <client>
        <!-- This entry only works with the RemotingHelper class -->
        <wellknown type="General.IRemoteCustomerManager, General"
                   url="http://localhost/MyServer/CustomerManager.rem" />
      </client>
    </application>
  </system.runtime.remoting>
</configuration>
```

Debugging in IIS

To debug your server application in IIS, you could follow the same approach presented earlier for the Windows service. You could select Debug ➤ Processes and manually attach the debugger to the ASP.NET worker process (aspnet_wp.exe or w3wp.exe, depending on the OS).

There is, however, an easier but not very obvious solution to debugging your remoting server inside of IIS. In essence, you have to "trick" Visual Studio .NET into attaching itself automatically to the ASP.NET worker process. To do this, you add a new ASP.NET Web Application Project to your solution. This project will host your remoting configuration information and server-side assemblies.

In this project, you can add the <system.runtime.remoting> section to your web.config file as discussed previously. To add server-side implementation and interface assemblies, you just have to reference them using the Add Reference dialog box. These will be automatically deployed to the correct subdirectory.

You must not delete the default WebForm1.aspx, as this will be used by Visual Studio to start the debugger. Now as soon as you hit F5, VS .NET will open an Internet Explorer window that displays this empty WebForm1.aspx. You can ignore this IE window, but you must not close it as this would stop the debugger. Instead, you can go to your server-side implementation code in VS .NET and simply set breakpoints, view variables, and so on as if you were running in a normal remoting host.

Summary

In this chapter, you learned about the different settings that can be employed in a configuration file. You now also know why configuration files are important and why you shouldn't hard code the connection information in your .NET Remoting clients.

You know how to use configuration files to allow for transparent use of configuration files for all types of remote objects. I also demonstrated a possible workaround for some problems that you might encounter when using [Serializable] objects in combination with SoapSuds-generated metadata. In the section "What About Interfaces?" in this chapter I also introduced an approach for using configuration files with an interface-based remote object, which eliminates the need for SoapSuds.exe.

I showed you different deployment scenarios, including managed applications such as a console application or a Windows service. You also read about the benefits of using IIS to host your remote objects and how to debug Windows services and IIS as remoting hosts.

In the next chapter, I show you how to build .NET Remoting clients for various technologies: Windows Forms, ASP.NET Web sites, and Web Services.

CHAPTER 5

Securing .NET Remoting

Unfortunately .NET Remoting does not incorporate a standard means of authentication. Instead, Microsoft opened the interfaces so that any authentication (and encryption) variant can be used by implementing custom sinks and sink providers. The effort to implement these sinks may be reasonable when developing large-scale distributed applications that have to integrate with already installed third-party sign-on solutions, but is definitely cumbersome, too.

In this chapter, I will go through the basic concepts for securing .NET Remoting components when hosted within Internet Information Services (IIS) as well as outside of IIS. You will learn how to enable authentication through IIS by leveraging basic HTTP sign-on as well as the more secure Windows-integrated authentication. Furthermore, you will see how to secure communication between the client and the server by enabling SSL at the server side.

Afterwards you will learn how to leverage additional components offered by Microsoft to secure .NET Remoting when not hosted within IIS, which is a little bit more complicated because you have to configure the security channels within the configuration.

Last but not least, you will learn about the security enhancements that will come with the .NET Framework 2.0 and Visual Studio 2005.

Building Secure Systems

Some of the major basic concepts for building secure systems are authentication, authorization, confidentiality, and integrity. Authentication and authorization are concepts that deal with an identity like a user or machine. Authentication is the process of determining who an actor is, and authorization is the process of permitting or denying access to resources to the actor.

When it comes to secure message exchange between two actors, you have to know about confidentiality and integrity. *Message confidentiality* means that a message sent between the client and the server cannot be read by anyone else but the sender and the receiver—the message can only be read by the client and the server and nobody else because it is encrypted with a key only known by those two actors. *Integrity* ensures that the messages do not change while being transmitted from the sender to the receiver. Techniques for ensuring integrity are message authentication codes and digital signatures.

But keep in mind, security is far more than just adding a couple of "security features" like encrypting messages. For example, while encryption ensures confidentiality, it does not protect you from replay attacks or denial of service attacks in any way. Furthermore, many security bugs can be application design issues at an architectural level (SSL does not solve all your problems).

Therefore, a very good understanding of the application's environment and potential threats in this environment is absolutely necessary. It's the only way of securing the architecture of your application. An example of a structured method for understanding the environment and potential threats is *threat modeling*.[1]

Authentication Protocols in Windows

In my opinion the most important authentication protocols that can be used in Windows are NTLM, Kerberos, and Negotiate. I want to give you a brief overview of how these protocols conceptually work so that you know the differences and you are able to select the appropriate authentication protocol for your solutions.

Although each of the authentication protocols works differently, they share one common thing: they require you to trust an authority that knows all the details about valid users and machines. When logging on to a machine using a local machine account, this authority is the local security authority subsystem (LSASS). If you log on to a domain using your domain user account, the domain controller acts as an authority who knows the details about the user account—in this case the actual authentication is performed by the domain controller and therefore the local machine has to trust the domain controller.

NTLM Authentication

NTLM, also known as *NT LAN manager authentication* or *Windows NT challenge/response*, is a very old authentication protocol that has been built into Windows since it has had networking support dating back to the good old LAN manager days. NTLM works only when both the client and the server are running on Windows. It implements a three-way authentication handshake between the client and the server as you can see in Figure 5-1.

1. For more information about threat modeling, take a look at Michael Howard and David LeBlanc's *Writing Secure Code, Second Edition* (Microsoft Press, 2002) or Frank Swiderski and Window Snyder's *Threat Modeling* (Microsoft Press, 2004).

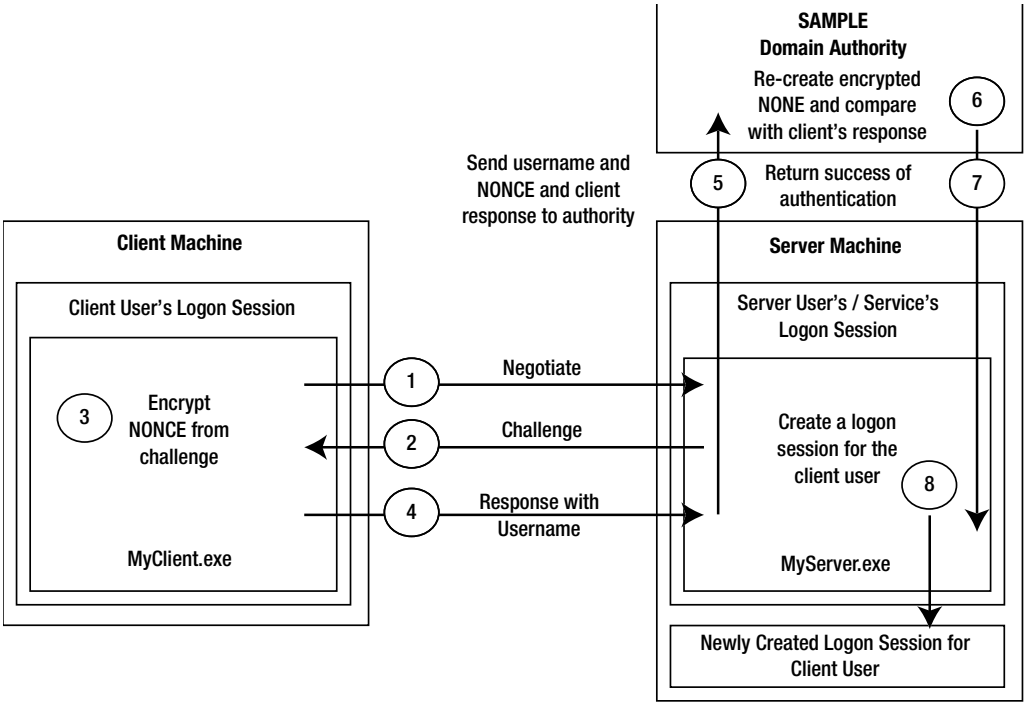

Figure 5-1. *NTLM authentication*

Both machines in the preceding diagram are in the same domain, SAMPLE. The client first starts the communication by sending a negotiate message to the server that contains just the username. The server responds with a challenge that consists of a NONCE—a random 64-bit value which is guaranteed to be unique for this particular communication session. When the client receives the challenge, it encrypts NONCE with the client user's master key (hashed password of the user), bundles it together with the principal name as well as the authority, and sends it back to the server.

Now the server can verify the client's credentials by sending the whole response as well as the original NONCE to the domain controller (remember, the server has generated the NONCE for the challenge, therefore it still knows it). The authority retrieves the hashed password from the security account manager database to again encrypt the clear text NONCE. If the result equals the encrypted NONCE sent in the client's response, authentication is successful and the server can create a network logon session for the client user on the server.

In case of an interactive logon where a user tries to access a server resource through a local account of the server, a logon screen will capture the username and password before sending the first negotiate message. Immediately after the username and password are entered Windows creates a one-way hash for the password (which will be used as the master key) and purges the original entries. This master key is used for encrypting the NONCE as described earlier.

Kerberos: Very Fast Track

Kerberos is a publicly well-known authentication protocol standardized by the IETF, and it is currently the most secure mechanism for authenticating users. It also has support for delegation, which means if the client allows it, the server can perform actions across the network on behalf of the client (although if possible avoid that because it always means some additional administrative overhead). On the Microsoft platform Kerberos has been available since Windows 2000. Covering Kerberos in detail would definitely require a book on its own, but I want to introduce the basic concepts briefly so that you understand the major differences.

The core of Kerberos consists of a set of *ticket-based* subprotocols that are used during the authentication process and, of course, an authority called the *Key Distribution Center* (KDC). Every actor within an authentication process has to trust the KDC (on Windows systems that is the domain controller). Kerberos requires the notion of a central authority and therefore can only be used in domains, while NTLM can be used for machine-to-machine communication, too.

As you can see in Figure 5-2, the client again wants to authenticate to the server. This time authentication is primarily based on tickets. Before the client can start a communication session with the server, it has to acquire a so-called *Ticket Granting Ticket* (TGT). Only if the client has a TGT can it request access to a server resource through the KDC. Therefore, to get a TGT the client has to authenticate at the KDC through the authentication service—basically the client provides his identity, and the KDC returns the TGT encrypted with the client's master key. That means only the valid client is able to decrypt the TGT. The TGT contains a session key used for further communication with the KDC and, of course, expires after a specific time defined by the KDC.

Next the client needs to request a ticket that permits access to the server resource it wants to access. Therefore it sends a request to the *Ticket Granting Service* (TGS). This request consists of the Ticket Granting Ticket received in the previous step, an authenticator, and the server the client wants to talk to. The authenticator is used to ensure the identity of the client requesting access to the server—it is nothing other than a time stamp encrypted with the client's master key (remember, the master key is the hashed version of the client user's password).

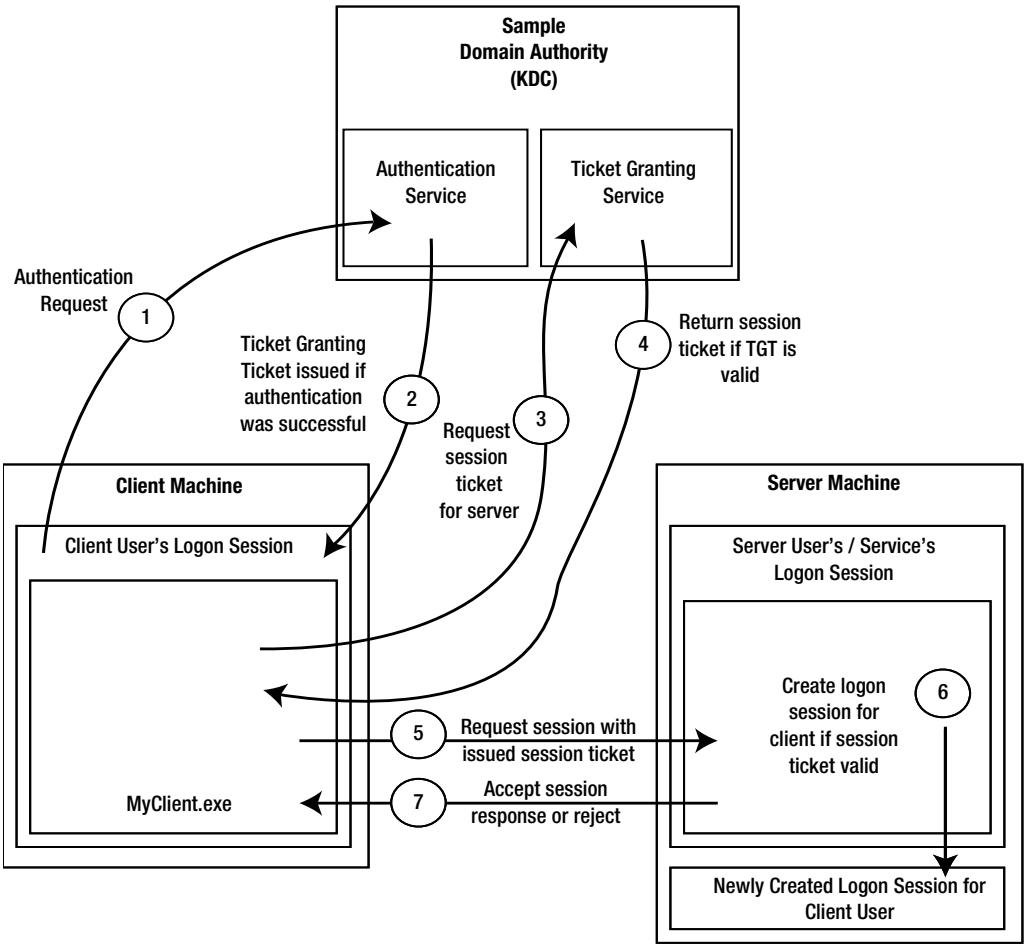

Figure 5-2. *Kerberos authentication actors*

The TGS responds with two parts. One part just contains the session key for the client encrypted with the client's master key and a *Session Ticket* (ST) encrypted with the server's master key that subsequently will be forwarded to the server. This ST also contains the session key as well as an expiration time and information about the client that wants to talk to the server.

When the server receives the ST, it decrypts it using its own master key (remember, the authority encrypted the ST using the server's master key). As the server and the KDC are the only parties that know the server's master key, they are the only ones that can decrypt the Session Ticket. Therefore, if decryption of the Session Ticket is successful on the server, the server knows that this ticket has been issued by a trusted authority (the KDC) and communication between the client and the server can start. Messages sent between the client and the server are encrypted using the session key issued by the TGS before (remember, the client got the session key as a response from its request to the TGS and the server found the session key in the Session Ticket received from the client).

Again, my intention in giving you this brief introduction was to outline the differences between NTLM and Kerberos so that you know what the prerequisites for using one of these protocols are. In general, you are always better off when using Kerberos because it is more secure, but as you have seen, this is not always possible (Windows 2000 and above required; active directory domain required). For more information about Kerberos version 5, take a look at `http://msdn.microsoft.com/library/default.asp?url=/library/en-us/secauthn/security/microsoft_kerberos.asp`.

Security Package Negotiate

The last authentication protocol I want to mention is the easiest one to understand—the security package negotiate (SPNEGO) authentication protocol. Basically this protocol is not doing the actual authentication itself, it just figures out the best (most secure) security package to use for the authentication process.

The idea of SPNEGO is fairly simple. Let's assume that both the client and the server select SPNEGO (negotiate) as their authentication protocol. SPNEGO always tries the most secure authentication package available. Therefore the client starts by selecting Kerberos and just sends the Kerberos authentication request across the wire to the server. If the server accepts the request, it directly responds with the corresponding Kerberos authentication reply (in this case the server accepts because SPNEGO has been selected on the server as well). If the server does not understand (or does not want to understand) the request, additional roundtrips are necessary for figuring out the next available security package (e.g., NTLM).

Security Support Provider Interface

As you have seen in the previous chapters, Windows supports a bunch of authentication protocols, and each of these protocols has its own specifics. Now imagine you want to develop an application that needs to support all of those protocols. Basically it means implementing a library for each authentication protocol that can be used from within the application, which is really cumbersome.

That's exactly where the *Security Support Provider* (SSP) and the *Security Support Provider Interface* (SSPI) become important. SSPI basically is an abstraction layer provided by Windows for encapsulating the specifics of authentication and secure conversation. As an application developer, you can appreciate how it enables you to authenticate a user across process and machine boundaries by just specifying the authentication protocol you want to use, and all the details of the protocol are handled by SSPI.

Unfortunately, SSPI does not exist as a managed API. But Microsoft provides a managed wrapper sample downloadable from MSDN (`http://msdn.microsoft.com/library/en-us/dndotnet/html/remsec.asp`) that encapsulates the Win32 SSPI API functions in a set of managed classes (see the section "Security Outside of IIS" later in this chapter).

Identities and Principals: A Short Overview

Before you can read about the details of securing .NET Remoting components, you have to understand how the .NET Framework supports implementing authentication and authorization.

Basically the .NET Framework differentiates between two types of security mechanisms: code access security (CAS) and role-based security (RBS). While code access security manages permissions on a code level by permitting or denying access to resources on a code basis, role-based security deals with user identification and authorization.[2]

The foundation for role-based security is laid by two interfaces that can be found in the System.Security.Principal namespace: IIdentity and IPrincipal. The identity can be seen as the result of an authentication process and identifies the user as well as the way the user has authenticated against the system. The principal is something like the account description for the user. Created with an identity, it connects the user with the roles assigned to the user. Based on the identity as well as the principal, you can implement your own authorization system. Out of the box the .NET Framework comes with four identities and two types of principals as you can see in Figure 5-3.

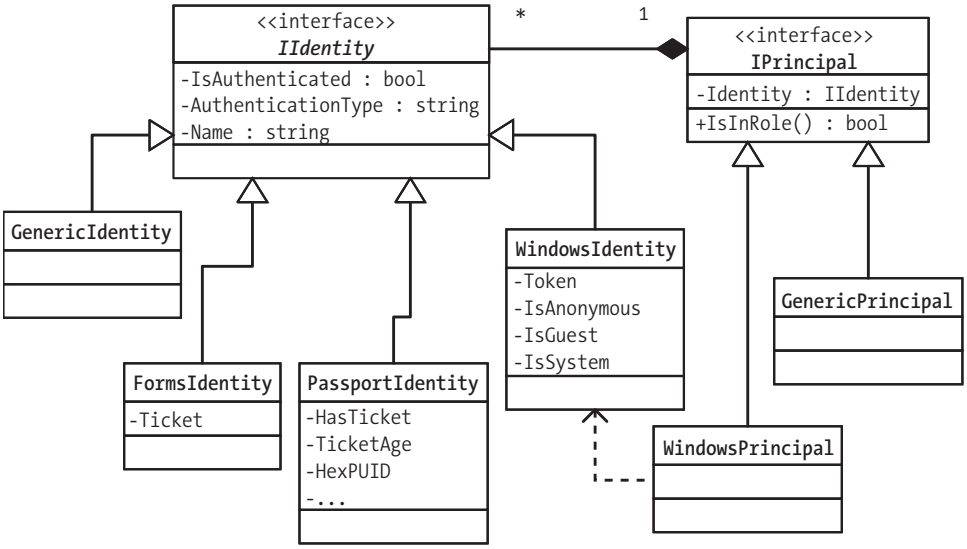

Figure 5-3. *IIdentity and IPrincipal*

2. Going into details of code access security would definitely require a book on its own. One of the best references for code access security I know is the book *.NET Framework Security* by Brian A. LaMacchia et al. (Addison-Wesley, 2002). For now, it is enough to know that code access security enables you to give permissions to code based on evidence like author, digital signature of the code, assembly origin, assembly name, or assembly version.

When working with identities and principals, you first need to authenticate the user with your preferred method. After authenticating successfully, you have to create an identity object based on the username and the authentication type. The principal object is then created based on the previously created identity object and optionally a list of roles.

The following table explains the differences between the types of identities shown in Figure 5-3.

Identity	Description
WindowsIdentity	WindowsIdentity is used for Windows authentication. This identity object is retrieved through a static method of the WindowsIdentity class called `GetCurrent()`. Therefore through `WindowsIdentity.GetCurrent()` you get an identity of the current process (the user account under which the process is running).
PassportIdentity	Passport is used for Passport authentication. ASP.NET supports passport authentication through configuration. Passport provides a single sign-on solution for Web-based applications.
FormsIdentity	If you configure your ASP.NET application using forms authentication and using the `System.Web.Security.FormsAuthentication.RedirectFromLoginPage()` method, a FormsIdentity will be created by the ASP.NET infrastructure for you and automatically assigned to HttpContext.Current.User.Identity.
GenericIdentity	You can use a GenericIdentity if none of the other identity objects fits into your solution. The GenericIdentity object has to be created manually with a username and authentication method as constructor parameters. Usually you create the GenericIdentity after you have authenticated the user through your own authentication mode (e.g., through a users table in your SQL Server database).

Note that you also can create your own identity by just creating a class and implementing the IIdentity interface. This might be interesting if you want to create your own identity object with additional functionality. The next table shows the different types of principals and their usage scenarios.

Principal	Description
WindowsPrincipal	WindowsPrincipal is used in conjunction with WindowsIdentity. Created based on a WindowsIdentity, it figures out the Windows groups for the WindowsIdentity and allows performing authorization based on the Windows groups.
GenericPrincipal	As its name implies, GenericPrincipal can be used in cases where you don't want to leverage the Windows infrastructure through the WindowsIdentity and WindowsPrincipal objects. A GenericPrincipal is created based on a previously created IIdentity and an array of strings that represent the roles for the user.

The following sample demonstrates the usage of WindowsIdentity and WindowsPrincipal in a simple console application project. It simply outputs the identity's name as well as the authentication type and whether the account is a guest account or not as you can see in Listing 5-1.

Listing 5-1. *A Simple Identity and Principal Sample*

```
using System;
using System.Security.Principal;

namespace WindowsIdentitySample
{
   class TestApplication
   {
      [STAThread]
      static void Main(string[] args)
      {
         // get the current windows identity and apply it to the managed thread
         WindowsIdentity identity = WindowsIdentity.GetCurrent();
         WindowsPrincipal principal = new WindowsPrincipal(identity);

         // output the identity's name as well as authentication method
         System.Console.WriteLine("User: " + identity.Name);
         System.Console.WriteLine("Authenticated through: " +
                                  identity.AuthenticationType);
         System.Console.WriteLine("Is Guest: " + identity.IsGuest);
      }
   }
}
```

If you take a look at the output of the application, it should resemble Figure 5-4.

Figure 5-4. *WindowsIdentity sample in action*

Of course, just identifying the user is not all you want to do. In the next step, you want to permit or deny access to application resources based on the user's identity or group and role membership. This can be achieved by simply querying the identity name or calling the `Principal.IsInRole(role name)` method. Modify the code from the previous sample so that it looks like Listing 5-2.

Listing 5-2. *The Modified Principal Excample with a Role Permission Check*

```
using System;
using System.Security.Principal;
using System.Security.Permissions;
```

```
namespace WindowsIdentitySample
{
   class TestApplication
   {
      [STAThread]
      static void Main(string[] args)
      {
         // get the current windows identity and apply it to the managed thread
         WindowsIdentity identity = WindowsIdentity.GetCurrent();
         WindowsPrincipal principal = new WindowsPrincipal(identity);

         // output the identity's name as well as authentication method
         System.Console.WriteLine("User: " + identity.Name);
         System.Console.WriteLine("Authenticated through: " +
                                  identity.AuthenticationType);
         System.Console.WriteLine("Is Guest: " + identity.IsGuest);

         // set the identity to the managed thread's CurrentPrincipal
         System.Threading.Thread.CurrentPrincipal = principal;

         // first of all demonstrate imperative security
         System.Console.WriteLine("\nTesting imperative security...");
         if(principal.IsInRole(@"BUILTIN\Administrators"))
         {
            System.Console.WriteLine(">> Administrative task performed <<");
         }
         else
         {
            System.Console.WriteLine("!! You are not an administrator !!");
         }

         // at last test declarative security
         try
         {
            System.Console.WriteLine("\nTesting declarative security...");
            DeclarativeSecurity();
            System.Console.WriteLine("Test succeeded!\n");
         }
         catch(System.Security.SecurityException ex)
         {
            System.Console.WriteLine("Security exception occured: " +
                                     ex.Message);
         }
      }

      [PrincipalPermission(SecurityAction.Demand, Role=@"BUILTIN\Users")]
      static void DeclarativeSecurity()
      {
```

```
            System.Console.WriteLine("Function called successfully!");
        }
    }
}
```

In the preceding sample, you can see one important fact: the .NET Framework role-based security distinguishes between two types of authorization checks! Authorization checks based on `Principal.IsInRole()` are called imperative security checks; you can also enforce security checks through the CLR itself by adding a permission demand attribute to the function as you can see with the DeclarativeSecurity function.

When using declarative security, the CLR itself enforces the permission as specified in the permission attribute. But to make declarative security work, the # for the executing managed thread must be initialized correctly (otherwise access is denied per default). When working with ASP.NET and IIS or the .NET Remoting security samples (see the section "Securing with IIS" in this chapter) the thread's `CurrentPrincipal` will be initialized automatically. In other applications you have to initialize the `CurrentPrincipal` yourself on each launched thread or call `AppDomain.CurrentDomain.SetThreadPrincipal(your principal)` to automatically let the CLR initialize the `CurrentPrincipal` for the threads of the AppDomain automatically.

■ **Note** For more information on identities and principals, refer to the MSDN documentation (`http://msdn.microsoft.com/library/default.asp?url=/library/en-us/cpguide/html/cpconRole-BasedSecurity.asp`).

Now that you know how the .NET Framework supports different security models, I can show you the details of securing .NET Remoting solutions. First of all, I dive into the authentication within IIS and outside of IIS, and last but not least I will leverage the knowledge of principals and identities to perform simple authorization tasks.

Securing with IIS

The easiest way for securing .NET Remoting components is hosting them in Internet Information Services, or IIS. IIS provides you with authentication against Windows accounts as well as transport-level security through SSL. It allows you to restrict your callers through IP address restrictions, too.

Authentication with IIS

The functionality of IIS is not tailored to remoting, but applies to all IIS applications, including static HTML pages and ASP.NET Web applications. Actually, .NET Remoting components are hosted within the ASP.NET runtime infrastructure. Because SSL will be configured through the IIS management console the same way as for any other IIS application without any effects on the code of your .NET Remoting application, the focus will be on authentication. The following table lists the authentication modes supported by IIS.

Authentication Mode	Description
Anonymous access	Configures the Web application so that the client does not need to authenticate to the server when doing requests. In this case, the application defaults to a user account called IUSR_MACHINENAME.
Basic authentication	Basic authentication is a standard authentication mode defined by the W3C. When using basic authentication, the client has to send user credentials in the HTTP header to the server. The server afterwards authenticates the user and grants or denies access based on the authenticated credentials. Basic authentication sends the username and password in clear text across the wire and therefore should be used only together with SSL.
Digest authentication	Digest authentication is a more secure authentication mode. It hashes the password before sending it across the wire. Therefore, the password is not sent clear text across the wire. However, a hacker can still try to break the password using brute force attacks or dictionary attacks.
Windows authentication	Windows authentication is the most secure authentication protocol available for IIS because it uses NTLM or, if possible, Kerberos authentication (see the section "Authentication Protocols in Windows" earlier in this chapter). Windows authentication is the best case for any intranet scenarios or extranet scenarios, but it does not work in any case for Internet scenarios because some proxies and firewalls block the authentication requests as well as necessary ports. Windows authentication has to be supported by the client, server, and any proxy server in between.

It doesn't matter which of these authentication modes you are using. Whether basic, digest, or Windows, for any user who wants to access the Web application or .NET Remoting component hosted in IIS, a valid Windows user account must be available (either a local machine or Active Directory account). That is different from, for example, ASP.NET forms authentication, which can be used with ASP.NET Web applications and provides developers with the possibility of validating credentials against a custom store like a SQL Server database.

The security properties can be specified in the IIS snap-in to the MMC. You can see a version of the previous server configured for authenticated use in Figure 5-5.

■ **Caution** The basic authentication scheme, which is usable for remoting components, is *not at all secure*. Using a TCP/IP packet sniffer, one can easily capture the authentication tokens and resend them with their own requests to "impersonate" the original user. For really secure applications, encryption using SSL is highly recommended, and may even be a necessity.

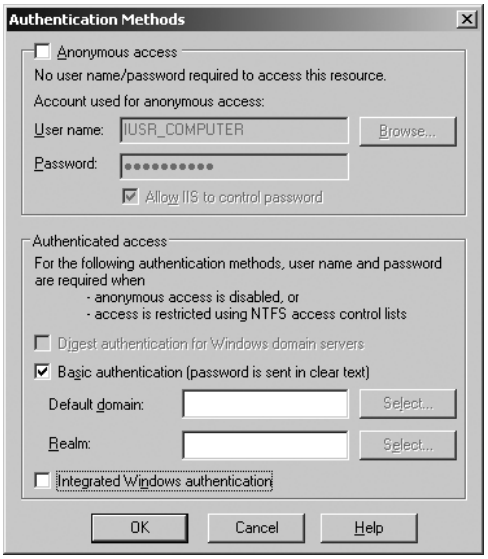

Figure 5-5. *Authentication is necessary when accessing remote objects.*

When using authentication within IIS, a change in the client code is necessary to send usernames and passwords to your server. Before running the following example, you have to create a new local user called DummyRemotingUser with the password 12345 on your system.

To transfer the logon information to the server, you have to set properties on the channel sink. Unfortunately, there is no direct way to specify a valid username/password combination for a given hostname (but I'll show you how to do this using a custom sink in Chapter 12). In the meantime, you have to call the static method `ChannelServices.GetChannelSinkProperties()`, which takes the proxy to the remote object as a parameter. This function returns an IDictionary that allows you to set extended properties including username and password.

```
IDictionary props = ChannelServices.GetChannelSinkProperties(mgr);
props["username"] = "dummyremotinguser";
props["password"] = "12345";
```

You should also extend the client to contain code to catch possible exceptions. These can occur due to misconfigurations on the server side or when passing an incorrect username/password combination. The complete source code for an authenticated client is shown in Listing 5-3.

Listing 5-3. *Client That Uses IIS's Built-In Authentication Methods*

```
using System;
using System.Runtime.Remoting;
using System.Runtime.Remoting.Channels;
using System.Collections;
using System.Runtime.Remoting.Services;
using General;  // from General.DLL
using Server;   // from server.cs

namespace Client
{
   class Client
   {
      static void Main(string[] args)
      {

         try
         {
            String filename = "client.exe.config";
            RemotingConfiguration.Configure(filename);

            CustomerManager mgr = new CustomerManager();

            Console.WriteLine("Client.Main(): Reference to CustomerManager " +
                              " acquired");

            IDictionary props = ChannelServices.GetChannelSinkProperties(mgr);
            props["username"] = "dummyremotinguser";
            props["password"] = "12345";

            Customer cust = mgr.getCustomer(4711);
            int age = cust.getAge();
            Console.WriteLine("Client.Main(): Customer {0} {1} is {2} " +
                              "years old.",
               cust.FirstName,
               cust.LastName,
               age);
         }
         catch (Exception e)
         {
            Console.WriteLine("EX: {0}",e.Message);
         }

         Console.ReadLine();
      }
   }
}
```

This client now connects to the server and authenticates the user against the specified Windows user account on the server machine. You can see in Figure 5-6 what happens when you change the password for DummyRemotingUser.

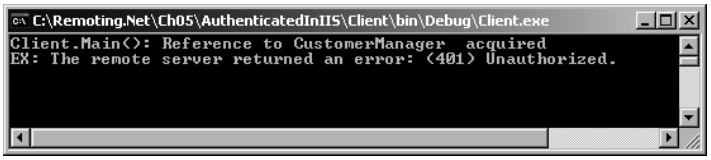
C:\Remoting.Net\Ch05\AuthenticatedInIIS\Client\bin\Debug\Client.exe
Client.Main(): Reference to CustomerManager acquired
EX: The remote server returned an error: (401) Unauthorized.

Figure 5-6. *Incorrect username/password combination*

Securing the Sign-On Process

In the preceding examples, I used the so-called HTTP basic authentication. This enables a great deal of interoperability between various Web servers and proxies. Unfortunately, this type of authentication allows a playback attack, which means that someone who uses software or a device to monitor all network traffic can later incorporate the transferred username/password combination in his or her own requests.

When both the client and the server are based on Windows XP, 2000, or NT, you can use Windows integrated authentication, which results in either NTLM or Kerberos (see "Authentication Protocols in Windows" earlier in this chapter). But remember that NTLM is a nonstandard mechanism that unfortunately does not work with all HTTP proxies. Furthermore, it requires you to open some ports on your firewall. However, if it is supported by the proxies of your users or your work in an intranet scenario, it nevertheless provides considerably higher security against playback attacks.

You can switch to this authentication scheme using Internet Services Manager MMC, as shown in Figure 5-7. Neither the client's code nor the server's code has to be changed after this switch.

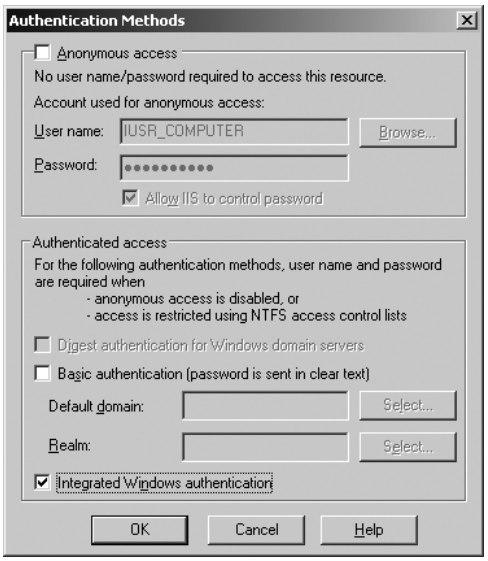

Figure 5-7. *Enabling Windows authentication*

■ **Note** You can also enable both basic and Windows authentication at the same time. The remoting framework (as well as standard Internet Explorer) will choose the most secure method that has been announced by the server.

Enabling Single Sign-On

When your user is authenticated against the same Windows domain in which your server is located, you finally can use integrated security. This will log your users on to the server without further need of specifying usernames or passwords.

The HTTP channel has a property called useDefaultCredentials. When this property is set to true via the configuration file and no username or password is specified within the ChannelSink's properties, the credentials of the currently logged-on user will be passed to the server. Because Windows 2000 can't get to a user's cleartext password, this scheme is only possible when using Windows authentication on your Web server.

When you want to switch to this authentication scheme, you just have to remove all calls to the channel sink's properties, which set the username or password, and instead include the following configuration file:

```
<configuration>
  <system.runtime.remoting>
    <application>
      <channels>
         <channel ref="http" useDefaultCredentials="true" />
      </channels>

      <client>
        <wellknown type="Server.CustomerManager, Client"
             url="http://localhost:8080/MyAuthServer/CustomerManager.soap" />
      </client>

    </application>
  </system.runtime.remoting>
</configuration>
```

Encryption and IIS

Using authentication, especially the Windows NT challenge/response authentication method, will give you a somewhat secured environment. Nevertheless, when transferring sensitive data over the Internet, authentication is just not enough—encryption needs to be applied as well.

Hosting your components in IIS gives you a head start when it comes to encryption, as you can easily leverage the built-in SSL capabilities. All it takes is installing a server-side

certificate[3] and changing the URL in the client-side configuration file. After making an edit to just one line (changing "http:" to "https:"), all traffic will be secured—including the HTTP headers, authentication information, and, of course, the transferred data.

The changed configuration file looks like this:

```
<configuration>
  <system.runtime.remoting>
    <application>
      <client>
        <wellknown type="Server.CustomerManager, Client"
                   url="https://localhost/MyAuthServer/CustomerManager.soap" />
      </client>
    </application>
  </system.runtime.remoting>
</configuration>
```

SSL encryption is sometimes accused of imposing a somewhat huge overhead. This is not always true, because the "real" asymmetric cryptography only takes place during the process of establishing the secured HTTP connection. This secure connection will be reused, and the overhead thus minimized.

■ **Note** You can get certificates either through buying them from a well-known authority like VeriSign or you can set up your own certificate authority (Windows 2000 Server or Windows Server 2003 includes certificate services that can be installed). When using your own certificate authority or using selfssl.exe of the IIS resource kit for issuing a server certificate, you have to configure the client to trust your certificate authority by installing the authority's certificate in the Trusted root Certification Authorities area in your Windows certificate store.[4]

When testing the example in Chapter 2 using both HTTPS and HTTP, you'll see that a binary formatter via HTTPS/SSL is *faster,* and fewer bytes are transferred over the network than when using a SOAP formatter via conventional HTTP.

---

3. Hint: you can get free certificates for development purposes from VeriSign (`http://www.verisign.com`). You can also download and install the IIS resource kit from `http://www.microsoft.com/downloads`, which includes a tool called selfssl.exe that can be used for creating a SSL certificate as well as enabling SSL on IIS with just one step. SelfSSL is intended for installing SSL on development and/or test machines.
4. To get to the Windows certificate store, log on as administrator and start a management console through Start ➤ Run ➤ mmc.exe. Afterwards select File ➤ Add/Remove Snap In and add the Certificates MMC snap-in. This snap-in allows you to manage the certificates of your local machine as well as the current user profile. Actually, certificates are stored in the Documents and Settings\All Users\Application Data\Microsoft directory.

Security Outside of IIS

Even though IIS provides a secure and scalable hosting environment for .NET applications, you might want or need to host your components in a custom Windows service. In this case, there is no built-in support for security in the .NET Remoting framework.

Of course, the first idea would be to manually serialize and transfer the user's credentials across the wire, but that is definitely insecure as the server has no way to figure out whether trusting the information received is a good idea or not. Furthermore, information is not encrypted so you would have to provide your own encryption and digital signature functionality, which leads to key exchange issues. (When and how to exchange keys? Where should you store the keys?)

Instead, there are other cryptographically safe methods for exchanging these credentials. Microsoft for example released an additional component in 2003 that provides for new security features for .NET Remoting. This component is now available in version 2.0 including complete source code and extensive documentation from `http://msdn.microsoft.com/library/en-us/dndotnet/html/remsec.asp`. To compile and use this component, you also have to download a managed wrapper of the SSPI features, which provide access to the underlying security infrastructure of the Windows operating system. You can download the SSPI component—also including source code and lots of documentation—from `http://msdn.microsoft.com/library/en-us/dndotnet/html/remsspi.asp`.

■ **Note** Please be aware that this component is not officially supported by Microsoft, the publisher of this book, or its authors. Version 2.0 of .NET will, however, include similar security functions that can be used independently of the chosen transport protocol.

Using the MSDN Security Samples

After downloading and extracting these components, you will find two new DLLs in subdirectories of the installation folders:

- Microsoft.Samples.Security.SSPI.DLL
- Microsoft.Samples.Runtime.Remoting.Security.DLL

The first component is a managed wrapper around the SSPI Win32 API functions, while the second component is a set of remoting extensibility classes (remoting sinks[5]) that leverages the SSPI wrapper for providing the security features to .NET Remoting.

For using the components provided with the sample, you have to complete two steps: first of all you have to add a reference to both assembly DLLs in the client as well as the server through the Visual Studio .NET Add Reference dialog box. In a second step, you have to configure the extensions through configuration files.

Let's start with implementing a sample that leverages the security samples provided by Microsoft. At first you will get a short look at the shared assembly that defines the serializable types that will be sent across the wire as well as the interface for the server-side singleton object as you can see in Listing 5-4.

---

5. I discuss these sinks and how to create them in Chapter 13.

Listing 5-4. *The Shared Assembly for the .NET Remoting Security Sample*

```
using System;

namespace General
{
   public interface IPersonFactory
   {
      Person GetPerson();
   }

    [Serializable]
   public class Person
   {
      private int _age;
      public string Firstname, Lastname;

      public Person(string firstname, string lastname, int age)
      {
         this.Age = age;
         this.Firstname = firstname;
         this.Lastname = lastname;
      }

      public int Age
      {
         get { return _age; }
         set
         {
            if(value >= 0)
               _age = value;
            else
               throw new ArgumentException("Age must be zero or positive!");
         }
      }
   }
}
```

Implementing the client is fairly easy as it does nothing other than retrieve a person from the server and show its content. For enabling security in the client, configuring the security sample provider correctly is the only thing you have to do. Listing 5-5 shows the client implementation.

Listing 5-5. *The Security Sample Client*

```
using System;
using System.Security.Principal;
using System.Runtime.Remoting;
using System.Runtime.Remoting.Activation;
```

```
using General;

namespace Client
{
   class ClientApp
   {
       [STAThread]
      static void Main(string[] args)
      {
         // startup information
         System.Console.WriteLine("Starting client...");
         System.Console.WriteLine("Current identity: " +
               WindowsIdentity.GetCurrent().Name);

         // configure the remoting runtime
         RemotingConfiguration.Configure(
            AppDomain.CurrentDomain.SetupInformation.ConfigurationFile);

         // instantiate the type
         try
         {
            IPersonFactory personCreator =
               RemotingHelper.CreateProxy(
                  typeof(IPersonFactory)) as IPersonFactory;
            Person p = personCreator.GetPerson();
            System.Console.WriteLine("Got the person: {0} {1} {2}",
                                     p.Firstname, p.Lastname, p.Age);
            System.Console.ReadLine();
         }
         catch(Exception ex)
         {
            System.Console.WriteLine("Exception occured: " + ex.Message);
         }

         System.Console.WriteLine("Press key to stop...");
         System.Console.Read();
      }
   }
}
```

The client at first retrieves the identity of its host process through `WindowsIdentity.GetCurrent()` and afterwards instantiates the proxy and calls the server. A little bit more interesting is the configuration for the client because of the additional configuration required for enabling security in the application. Don't forget to add a reference to the security sample DLLs through the Add Reference dialog box of Visual Studio .NET.

```
<configuration>
<system.runtime.remoting>
  <application>
    <channels>
      <channel ref="tcp">
        <clientProviders>
          <formatter ref="binary" />
          <provider type="Microsoft.Samples.Runtime.
                  Remoting.Security.SecurityClientChannelSinkProvider,
                  Microsoft.Samples.Runtime.Remoting.Security"
              securityPackage="negotiate"
              impersonationLevel="identify"
              authenticationLevel="packetPrivacy" />
        </clientProviders>
      </channel>
    </channels>
    <client>
      <wellknown type="General.IPersonFactory, General"
                 url="tcp://localhost:1234/MyPersonManager.rem" />
    </client>
  </application>
</system.runtime.remoting>
</configuration>
```

Within the clientProviders section of the configuration you add the security provider immediately after the binary formatter. The security provider supports a set of additional properties that you can see in the following table.

Setting	Values	Description
securityPackage	ntlm, kerberos, negotiate	*Client and server*; specifies the security protocol that will be used for the authentication handshake. Currently only NTLM, Kerberos, or SPNEGO are supported by this implementation.
impersonationLevel	identify, impersonate, delegate	*Client only*; specifies the capabilities of the security token passed over to the server. The option *identify* means that the server is able to identify the client while the option *impersonate* allows the server to identify as well as impersonate the security token of the client to act on behalf of the client on the local machine. Finally the option *delegate* makes it possible to identify the client, impersonate the security token, as well as delegate the token to act on behalf of the client across network hops.

Continues

Setting	Values	Description
authenticationLevel	call, packetIntegrity, packetPrivacy	*Client and server*; essentially the authentication level specifies the level of protection for messages being exchanged between the client and the server (therefore I think that the name is a little bit misleading): *call* means that the client gets authenticated on each method call. Messages are sent unprotected across the wire. With the option *packetIntegrity* the client gets authenticated on each method call and the sender adds a digital signature to the message. On the receiver's side the digital signature will be verified. The last option, *packetPrivacy*, means that the client gets authenticated on each method call and the message will be encrypted by the sender. The receiver decrypts and subsequently processes the message. The reason why you can select these different levels is that you have to make a performance vs. security decision. The more secure your application becomes, the more processing power is needed.

The Kerberos security package can only be used when working within a domain and having a connection to the domain controller (KDC). Otherwise, NTLM must be used (you can also use negotiate, which will result in NTLM in this case and in Kerberos if you are working within the domain having a connection to the KDC).

■ **Note** Windows 2000 and above supports cached credentials, which allow you to log on to your machine with your Active Directory domain account even if you don't have a connection to the KDC (on Windows this is the domain controller). If you are testing or developing .NET Remoting components under your domain user account using the SSPI security solution, you will not be able to impersonate credentials from your client (running under your domain user account) to a .NET Remoting server running in a different logon session under a different user account because you either don't have a ticket for impersonation or the ticket is expired.

The authentication level must be set on both the client and the server. But what happens if the level on the client does not match the level on the server? Similar to DCOM, the behavior is as follows: if the authentication level on the client is lower than the authentication level on the server, then an exception will be thrown. If the authentication level on the client is higher than on the server, the server will switch to the higher authentication level. Of course, the lowest authentication level is call and the highest authentication level is packetPrivacy.

Essentially, this means that a minimum authentication level can be enforced by the server. If the client exceeds this level, the server is okay; if not, it rejects the request and throws an exception. For example, if the server is configured for packetIntegrity and the client for call, then the server will not accept the message and throws an exception. On the other hand, if the client is configured to packetPrivacy, in this case the server switches to the higher authentication level and also uses packetPrivacy.

Now let's switch to the implementation of the server that will only accept authenticated requests. The server outputs information about the client user calling the server as well as the identity of the process in which the server is hosted. Listing 5-6 shows the complete implementation of the server.

Listing 5-6. *The Server for the Security Sample*

```
using System;
using System.Security.Principal;
using System.Runtime.Remoting;

namespace Server
{
   public class PersonManager : MarshalByRefObject, General.IPersonFactory
   {
      public General.Person GetPerson()
      {
         try
         {
            WindowsIdentity identity = WindowsIdentity.GetCurrent();

            System.Console.WriteLine("\nIncoming request...");
            System.Console.WriteLine("Current windows identity: " +
                                                    identity.Name);
            System.Console.WriteLine("Current thread identity: " +
                     System.Threading.Thread.CurrentPrincipal.Identity.Name);
         }
         catch(Exception ex)
         {
            System.Console.WriteLine("Exception occured: " + ex.Message);
         }

         System.Console.WriteLine("Returning a new person...");
         return new General.Person("Mini", "Coperground", 50);
      }
   }

   public class ServerApp
   {
      [STAThread]
      static void Main(string[] args)
      {
         string configFile =
            AppDomain.CurrentDomain.SetupInformation.ConfigurationFile;

         RemotingConfiguration.Configure(configFile);
         System.Console.WriteLine("Server started, waiting for requests...");
         System.Console.WriteLine("Server's process user: " +
                                  WindowsIdentity.GetCurrent().Name);
```

```
            System.Console.ReadLine();
        }
    }
}
```

Essentially the server is a Singleton object that acts as a factory for the person object. When a client requests a new person, the server retrieves the current identity of the application's host process through `WindowsIdentity.GetCurrent()`. Afterwards it retrieves the principal identity of the managed thread that will be set by the security solution components and identifies the client credentials. Just take a look at Figure 5-8 to see what happens if the server is called by a client running in a different logon session.

Figure 5-8. *The server sample in action*

■ **Note** If you want to run the client as well as the server in different logon sessions, you can use the runas.exe command to start either the client or the server or Visual Studio .NET in a different logon session without logging on and off from your machine.

Next, take a look at the configuration for the server, which looks similar to the client's configuration:

```
<configuration>
  <system.runtime.remoting>
    <application>
      <channels>
        <channel ref="tcp" port="1234">
          <serverProviders>
            <provider type="Microsoft.Samples.Runtime.
                      Remoting.Security.SecurityServerChannelSinkProvider,
                      Microsoft.Samples.Runtime.Remoting.Security"
                  securityPackage="negotiate"
                  authenticationLevel="packetPrivacy" />
            <formatter ref="binary" />
          </serverProviders>
        </channel>
```

```
      </channels>
      <service>
        <wellknown type="Server.PersonManager, Server"
                   objectUri="MyPersonManager.rem"
                   mode="Singleton" />
      </service>
    </application>
  </system.runtime.remoting>
</configuration>
```

The only difference between the configuration of the client and the configuration of the server is that the serverProviders configuration option is used this time, and the SecurityServerChannelSinkProvider is used instead of the SecurityClientChannelSinkProvider. Also the formatter is specified after the security provider to verify security before deserializing the message.

■ **Note** The impersonationLevel option is not used by the server as it gives the client the capability to specify the capabilities of the token sent from to the server. As only the client should be able to specify the capabilities of the security token sent to the server, this option is not useful for the server at all.

Specifying an impersonationLevel of impersonate or delegate allows the client to specify the capabilities of the security token sent to the server. The server authenticates the user but takes no further action—this means it does not impersonate the caller automatically. To verify and test this, append the following code within the try block of the `GetPerson()` method of the PersonManager class in the server:

```
// get the client's identity and impersonate the token
identity = System.Threading.Thread.CurrentPrincipal.Identity as WindowsIdentity;
WindowsImpersonationContext impCtx = identity.Impersonate();

// now output the current windows identity
System.Console.WriteLine("Identity impersonated...");
System.Console.WriteLine("Current identity: {0}",
                            WindowsIdentity.GetCurrent().Name);

// revert the identity to itself and again output the current windows identity
impCtx.Undo();
System.Console.WriteLine("Identity reverted: {0}",
                            WindowsIdentity.GetCurrent().Name);
```

Now let's test the application again and take a look at the output of the server. Notice that you originally configured the impersonationLevel in the first version of the client to the setting `identify`, which only allows the server to identify the client but not impersonate or delegate the security token. Therefore you are running in an exception this time, as you can see in Figure 5-9.

cmd (running as THINKTECTURE\demouser) - Server.exe
Server started, waiting for requests...
Server's process user: THINKTECTURE\demouser
Incoming request...
Current windows identity: THINKTECTURE\demouser
Current thread identity: THINKTECTURE\ingo
Identity impersonated...
Exception occured: Access is denied.

Figure 5-9. *Impersonation error*

In the next step, change the impersonationLevel setting on the client to impersonate and run the sample again. This time the server is able to impersonate the client as you can see in the output shown in Figure 5-10.

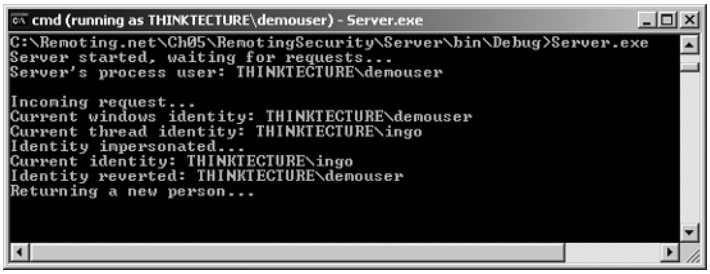

Figure 5-10. *Successful impersonation of the client*

With each incoming request, the server outputs its current Windows identity (which is the identity of the host process), and afterwards the identity of the managed thread, which is the identity passed from the client to the server. Without impersonating, the process would access external resources like files or a database through the process identity, which means that access control is verified against the process identity, too. If the server impersonates the client, then the identity will be applied to the unmanaged thread, which means that each access to external resources happens with the client's identity. Therefore, in the case of file system ACLs or in the case of the database, logins must be configured to permit access for the client's identity.

■ **Note** To really see the difference in impersonating the client on the server side, you should run the client and the server under different user accounts. You can do that by either running Visual Studio .NET or the applications in a separate login session using the runas command.

Finally, although not supported by Microsoft or the authors of this book, it makes sense to use the security samples for .NET Remoting. Also, most of the code you are writing when using these samples is still valid with Visual Studio .NET 2005 and .NET Framework 2.0, as you will see in the next chapters.

Implementing Authorization in the Server

Now that you know how to authenticate the user, you can implement authorization based in the user's identity as well as the principal. The solutions introduced in the previous chapters, the IIS as well as the MSDN security sample solution, automatically initialize the `System.Threading.Thread.CurrentPrincipal` property with a WindowsPrincipal of the authenticated user. That enables you to allow only certain users to use your service or even assign different privileges to different users.

■ **Tip** Be sure to pass not only the name of the group but also the name of your machine or domain, as in IsInRole(@"YOURMACHINE\ThisGroup") or IsInRole(@"YOURDOMAIN\ThisGroup"). You can get the name of the current computer from Environment.MachineName.

You accomplish the security by just retrieving the `CurrentPrincipal` and verifying group memberships for the user.

■ **Caution** The standard group names are not language agnostic. For example, on a German version of Windows the administrators group is called "YOURMACHINE\Administratoren", while on an English version of Windows it is called "YOURMACHINE\Administrators". Even the notation "BUILTIN\groupname", which can be used for standard Windows groups (like Administrators or Power Users), is not language agnostic and also reflects changes of the group names (for example, if you rename the Administrators group to Admins, then you have to use BUILTIN\Admins in your application, too). The only way for doing language and name-change-safe verifications on Windows groups is through the WindowsBuiltInRole enumeration.

■ **Tip** For custom application authorization, I'd suggest defining application-specific roles and map any Windows accounts or Windows groups to those roles in your application data store (of course, assuming you want to use Windows authentication). This enables decoupling the application from the local Windows account and domain account as well as group structures and makes the application more reusable.

In Listing 5-7, I show you how to extend the server to check whether the remote user is in the group RemotingUsers.

■ **Note** You can create a group and assign users to it using the MMC (access this by right-clicking My Computer and selecting Manage ➤ System Tools ➤ Local Users and Groups).

Listing 5-7. *Checking the Membership in Windows Groups When Hosting in IIS*

```
using System;
using General;
using System.Security.Principal;
```

```
namespace Server
{
   class CustomerManager: MarshalByRefObject
   {
      public CustomerManager()
      {
         Console.WriteLine("CustomerManager.constructor: Object created");
      }

      public Customer getCustomer(int id)
      {
         String machinename = Environment.MachineName;

         IPrincipal principal =
               System.Threading.Thread.CurrentPrincipal;

         if (! principal.IsInRole(machinename + @"\RemotingUsers"))
         {
            throw new UnauthorizedAccessException(
                        "The user is not in group RemotingUsers");
         }

         Console.WriteLine("CustomerManager.getCustomer): Called");
         Customer tmp = new Customer();
         tmp.FirstName = "John";
         tmp.LastName = "Doe";
         tmp.DateOfBirth = new DateTime(1970,7,4);
         Console.WriteLine("CustomerManager.getCustomer(): Returning " +
            "Customer-Object");
         return tmp;
      }
   }

}
```

You can use the same client as in the earlier example with this server. Depending on the group membership of the user, you will see the output in Figure 5-11 when DummyRemotingUser is a member of RemotingUsers, or the output in Figure 5-12 when DummyRemotingUser is not a member of this group.

■ **Note** Updates to group membership or users' passwords are not reflected online in IIS. You might have to either wait a little or restart IIS before such changes will take effect.

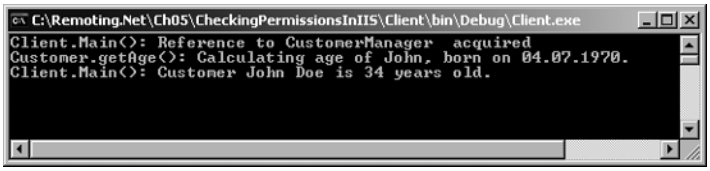
C:\Remoting.Net\Ch05\CheckingPermissionsInIIS\Client\bin\Debug\Client.exe
Client.Main(): Reference to CustomerManager acquired
Customer.getAge(): Calculating age of John, born on 04.07.1970.
Client.Main(): Customer John Doe is 34 years old.

Figure 5-11. *The user is in the correct group.*

C:\Remoting.Net\Ch05\CheckingPermissionsInIIS\Client\bin\Debug\Client.exe
Client.Main(): Reference to CustomerManager acquired
EX: The user is not in group RemotingUsers

Figure 5-12. *The user is not a member of the required group.*

Security with Remoting in .NET 2.0 (Beta)

With the next version of the .NET Framework, version 2.0 (codename Whidbey), security is an integral part of the .NET Remoting framework. The security infrastructure for .NET Remoting 2.0 includes authentication as well as channel security similar to the SSPI sample solution described earlier in this chapter.

Security in .NET Remoting 2.0 will be configured directly for the channel. Therefore, all you have to know about for leveraging the security infrastructure is a couple of new channel properties as well as enumerations used in conjunction with those properties. Take a look at the following table for an overview of the new properties and their associated enumerations.[6]

Property	Enumeration	Description
impersonationLevel	ClientImpersonationLevel	*Client only*; the impersonation level gives the client the possibility to specify the capabilities of the security token sent to the server. This allows the client to define whether the server is allowed to identify, impersonate, or delegate the client's security token. Possible values are Identify, Impersonate, Delegate, or None.
authenticationMode	AuthenticationMode	*Server only*; the authentication mode is used by the server to define the capabilities the security token sent from the client to the server must have so that the server can perform its work properly. If the capabilities of the client token do not match the requirements specified in this option, an exception will be thrown by the server.

Continues

6. All the security-related enumerations are defined in the System.Runtime.Remoting.Channels namespace.

Property	Enumeration	Description
encryption	Encryption	*Client and server*; the encryption option specifies the transport-level security for the messages sent between the client and the server. Both the client and the server must be configured with the same encryption settings. Possible options are None, Sign, and EncryptAndSign.

The first major difference between the SSPI security solution sample for .NET Remoting 1.x and the security infrastructure of .NET Remoting 2.0 is the fact that the server can specify the requirements for the client security token so that these requirements can be verified up front and not at the moment when the server, for example, tries to impersonate the client (as is the case with the SSPI sample solution).

Secondly, with .NET 2.0 the server automatically impersonates the client's identity if both the server and the client are configured for impersonation. You can see this when retrieving the Windows identity through `WindowsIdentity.GetCurrent()` without calling `WindowsIdentity.Impersonate()` up front.

Next you'll see a code sample that will give you a closer look at the new .NET Remoting infrastructure. You will develop a client as well as a server but will configure the client's channel programmatically and the server through the configuration file so that you can see both ways. The server of the sample offers a simple method that takes a string message as well as an integer counter. For this purpose, you'll define the server's interface in a shared assembly as you can see in the following code snippet:

```
namespace RemotedType
{
    public interface IRemotedType
    {
        void DoCall(string message, int counter);
    }
}
```

Listing 5-8 contains the server's implementation. The server offers a simple remote object that implements the interface. For each request the server outputs the authenticated user as well as the managed and the unmanaged thread's identity.

Listing 5-8. *A Server Using the .NET Remoting 2.0 Security Infrastructure*

```
using System;
using System.Collections.Generic;
using System.Runtime.Remoting;
using System.Runtime.Remoting.Messaging;

using System.Security;
using System.Security.Principal;

namespace RemotingServer
{
```

```
public class RemoteServerApp
{
    static void Main(string[] args)
    {
        try
        {
            System.Console.WriteLine("Configuring server...");
            System.Runtime.Remoting.RemotingConfiguration.Configure(
                                                "RemotingServer.exe.config");
            System.Console.WriteLine(
                                "Server configured, waiting for requests...");
            System.Console.ReadLine();
        }
        catch (Exception ex)
        {
            System.Console.WriteLine("Error while configuring server!");
            System.Console.ReadLine();
        }
    }
}

public class MyRemoteObject : MarshalByRefObject, RemotedType.IRemotedType
{
    public void DoCall(string message, int counter)
    {
        // get some information about the caller's context
        IIdentity remoteIdentity = CallContext.GetData(
                                    "__remotePrincipal") as IIdentity;
        if (remoteIdentity != null)
        {
            System.Console.WriteLine("Authenticated user:\n-){0}\n-){1}",
                                        remoteIdentity.Name,
                                remoteIdentity.AuthenticationType.ToString());

            // is the principal set on the managed thread?
            IIdentity threadId =
                        System.Threading.Thread.CurrentPrincipal.Identity;
            System.Console.WriteLine(
                    "Current threads identity: {0}!", threadId.Name);

            // get the identity of the process
            WindowsIdentity procId = WindowsIdentity.GetCurrent();
            System.Console.WriteLine("Process-Identity: {0}", procId.Name);
        }
        else
        {
```

```
                System.Console.Write("!! Attention, not authenticated !!");
            }

            // just do the work
            for (int i = 0; i < counter; i++)
            {
                System.Console.WriteLine("You told me to say {0}: {1}!",
                                                counter.ToString(), message);
            }
        }
    }
}
```

All you have to do for enabling security on the server is configure the `authenticationMode` setting for the server channel. Optionally you can configure the `encryption` setting for ensuring transport-level security.

```
<configuration>
    <system.runtime.remoting>
        <application name="MyServer">
            <service>
                <wellknown
                            type="RemotingServer.MyRemoteObject, RemotingServer"
                            objectUri="MyObject.rem"
                            mode="SingleCall" />
            </service>
            <channels>
                <channel ref="tcp"
                                port="9001"
                                encryption="None"
                                authenticationMode="IdentifyCallers" />
            </channels>
        </application>
    </system.runtime.remoting>
</configuration>
```

In this case the server requires the client to authenticate itself. Encryption is not required. To encrypt the traffic or to sign the messages sent across the wire, just specify the encryption option Sign or EncryptAndSign. For other authentication modes, configure the server with one of the options available in the AuthenticationMode enumeration.

Before you start changing the configuration, you create the client and test the solution with authentication through the TCP channel. This example demonstrates using the TCP channel instead of the IPC channel so that network traffic can be traced to see the effect of the options on the wire. In the client you configure the security options programmatically as you can see in Listing 5-9.

Listing 5-9. *A Client Using Code to Configure Security with .NET Remoting 2.0*

```
using System;
using System.Collections.Generic;
using System.Runtime.Remoting;
using System.Runtime.Remoting.Channels;
using System.Runtime.Remoting.Channels.Tcp;
using System.Runtime.Remoting.Activation;
using System.Text;

using RemotedType;

namespace RemotingClient
{
    class Program
    {
        static void Main(string[] args)
        {
            try
            {
                Dictionary<string, string> dict =
                          new Dictionary<string, string>();
                dict.Add("impersonationLevel", "Identify");
                dict.Add("encryption", "None");

                System.Console.WriteLine("Configuring channel...");
                TcpClientChannel clientChannel =
                                  new TcpClientChannel(dict, null);
                ChannelServices.RegisterChannel(clientChannel);

                System.Console.WriteLine("Configuring remote object...");
                IRemotedType TheObject = (IRemotedType) Activator.GetObject(
                    typeof(RemotedType.IRemotedType),
                    "tcp://localhost:9001/MyObject.rem");

                System.Console.WriteLine(
                         "Please enter data, 'exit' quits the program!");
                int c = 0;
                string input = string.Empty;
                do
                {
                    System.Console.Write("Enter message: ");
                    input = System.Console.ReadLine();
                    if (string.Compare(input, "exit", true) != 0)
                    {
                        System.Console.Write("Enter counter: ");
                        c = Int32.Parse(System.Console.ReadLine());
```

```
                    TheObject.DoCall(input, c);
                }
            } while (string.Compare(input, "exit", true) != 0);
        }
        catch (Exception ex)
        {
            System.Console.WriteLine("Exception while processing contents!");
            System.Console.ReadLine();
        }
    }
  }
}
```

The security configuration is done through channel properties, as you can see in the first lines of the preceding code. In this case, the client is configured to identify when calling the server and to not use any transport security through encryption or signature.

Figures 5-13 and 5-14 show the client and server in action. Notice that the server outputs the client's identity as well as the process identity. The server can retrieve the client identity in two ways: through the thread's `CurrentPrincipal` property or through the `__remotePrincipal` property of the CallContext data collection.

Figure 5-13. *The .NET Remoting 2.0–based client*

cmd (running as THINKTECTURE\demouser) - RemotingServer.exe
C:\Remoting.net\Ch05\WhidbeySecurity\RemotingServer\bin\Debug>RemotingServer.exe
Configuring server...
Server configured, waiting for requests...
Authenticated user:
->THINKTECTURE\ingo
->NTLM
Current threads identity: !
Process-Identity: THINKTECTURE\demouser
You told me to say 3: hello world!
You told me to say 3: hello world!
You told me to say 3: hello world!

Figure 5-14. *The .NET Remoting 2.0–based server*

■ **Note** As you can see in Figure 5-14, the server outputs nothing for the managed thread's identity. This means the .NET Remoting security infrastructure does not automatically initialize the managed thread's `CurrentPrincipal` property. You have to read the principal from the CallContext's `__remotePrincipal` property and manually assign this principal to System.Threading.Thread.CurrentPrincipal. But as you will see later, when the server is configured for impersonation, the thread's `CurrentPrincipal` will be automatically initialized.

The reason for selecting the TCP and not the (faster) IPC channel for this sample was to get an easy means for tracing the traffic between the client and the server to see how encryption works. Before changing any settings, let's use a trace tool for sniffing the traffic between the two applications.

I have selected the SOAP Trace Utility included in the Microsoft SOAP Toolkit 3.0 because it makes it really very easy to set up a tracing environment that works on the local machine. The SOAP Trace Utility acts as an intermediary proxy between the client and the server. It captures requests on a specific port, displays them, and forwards them to another port.

Therefore, when starting a trace, the trace utility asks you on which port to listen and to which port the requests should be forwarded. In this case, configure the tool for listening on port 8080 and forwarding to your server's port 9001. Furthermore, you have to use an unformatted trace because you are not using the SOAP formatter. The client's code must be modified so that the client sends the request to port 8080 instead of 9001 as you can see in the following code snippet:

```
System.Console.WriteLine("Configuring remote object...");
IRemotedType TheObject = (IRemotedType) Activator.GetObject(
    typeof(RemotedType.IRemotedType),
    "tcp://localhost:8080/MyObject.rem");
```

In Figure 5-15 you can see a screen shot of the running SOAP Toolkit sniffing the traffic between the client and the server. Take a close look at the text representation of the trace on the right side where you can see the clear text of your message ("hello world").

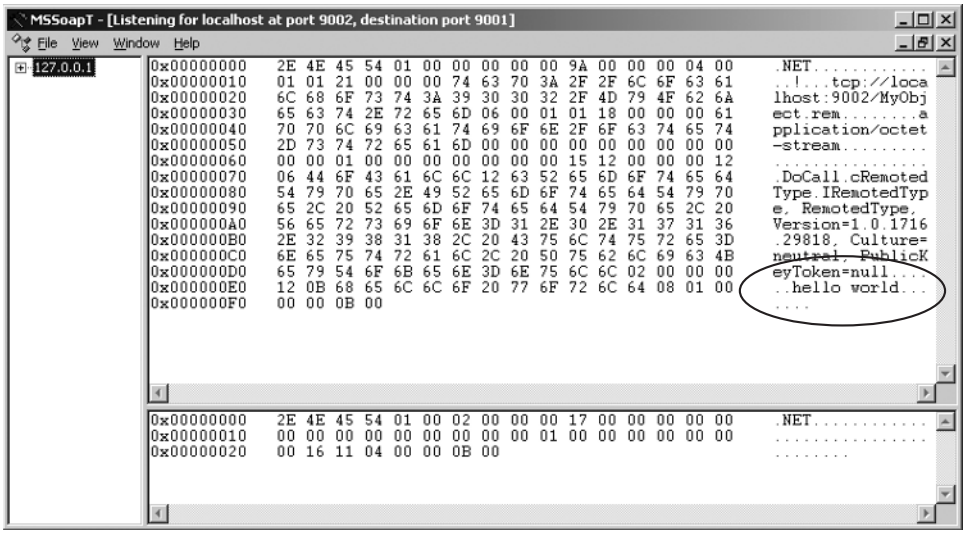

Figure 5-15. *Network sniffing of the unencrypted versions of the applications*

Now test the behavior of your applications when changing some of the security-related properties. First of all, change the client's encryption setting to EncryptAndSign. After recompiling the client and testing it again (see Figure 5-16), the call to the server will fail, as you can see in Figure 5-17 (the server doesn't even start processing the request).

file:///C:/Remoting.net/Ch05/WhidbeySecurity/RemotingClient/bin/Debug/RemotingClient.exe
Configuring channel...
Configuring remote object...
Please enter data, 'exit' quits the program!
Enter message: hello world!
Enter counter: 3
Exception while processing contents: Authentication has failed on the remote sid
e (stream may still be available for retrying).

Figure 5-16. *The client after the encryption setting changes—authentication failed*

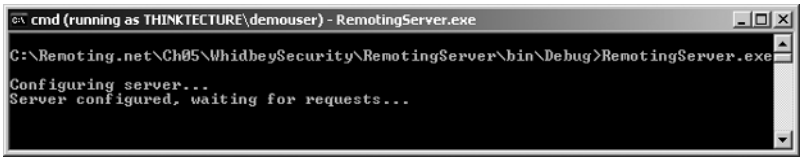

Figure 5-17. *Application logic of the server is not touched due to failed authentication.*

On the other hand, if the client is configured for *no encryption* (encryption="None") and the server requires encryption, the call fails, and the server returns an exception telling the client that security requirements are not fulfilled, as you can see in Figure 5-18.

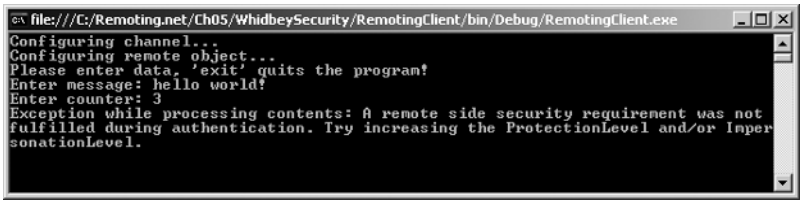
file:///C:/Remoting.net/Ch05/WhidbeySecurity/RemotingClient/bin/Debug/RemotingClient.exe
Configuring channel...
Configuring remote object...
Please enter data, 'exit' quits the program!
Enter message: hello world!
Enter counter: 3
Exception while processing contents: A remote side security requirement was not
fulfilled during authentication. Try increasing the ProtectionLevel and/or Imper
sonationLevel.

Figure 5-18. *The client does not fullfill the security requirements for the server.*

If you configure both the client and the server for the same encryption option, communication succeeds. To see the difference to the original configuration of the applications, take a look at the encrypted network trace in Figure 5-19—you can't see the "hello world" data string anymore.

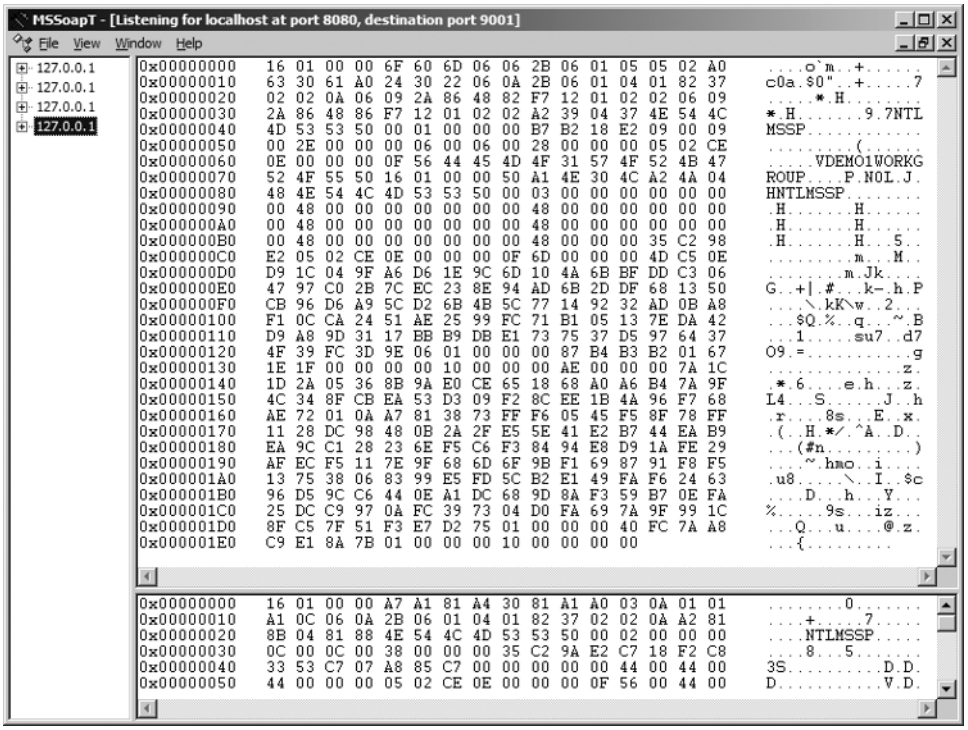

Figure 5-19. *Sniffing the encrypted network traffic*

Next, change the configuration settings for the impersonation level. If you change the server setting for the `authenticationMode` to ImpersonateCallers the server tries to impersonate each client. If the client's token does not allow the server to impersonate or delegate the security token[7] the server rejects the call and you get an exception on the client. If the client and the server are configured properly, impersonation (or delegation) works. To test delegation on a single machine, run the client under a different identity using the runas.exe command. Figures 5-20 and 5-21 show the two applications in action when the client's `impersonationLevel` property is set to impersonate and the server's `authenticationMode` is set to ImpersonateCallers.

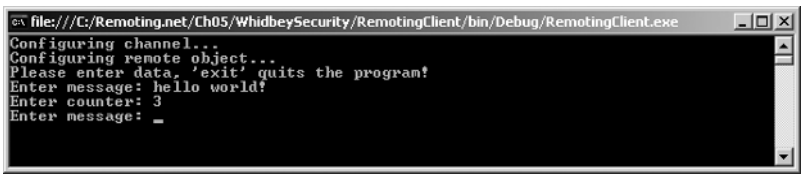

Figure 5-20. *The client runs under the current user's account.*

7. The `impersonationLevel` option on the client must be configured to either impersonate or delegate to allow the server to impersonate or delegate the client's security token.

cmd (running as THINKTECTURE\demouser) - RemotingServer.exe
C:\Remoting.net\Ch05\WhidbeySecurity\RemotingServer\bin\Debug>RemotingServer.exe
Configuring server...
Server configured, waiting for requests...
Authenticated user:
->THINKTECTURE\ingo
->NTLM
Current threads identity: THINKTECTURE\ingo!
Process-Identity: THINKTECTURE\ingo
You told me to say 3: hello world!!
You told me to say 3: hello world!!
You told me to say 3: hello world!!

Figure 5-21. *The server runs as DemoUser and impersonates the client.*

If you take a close look at the server's output, you can see that it automatically impersonates the client (remember, the server outputs the current unmanaged thread identity through `WindowsIdentity.GetCurrent()`). Also notice that the client is running under a different identity (take a look at the title of the command window).

Finally, the security classes in .NET Remoting 2.0 have all the features required for building secure .NET Remoting components based on the Windows security infrastructure and are easier to use than the SSPI sample for .NET Remoting 1.0, too.

Summary

In this chapter, I showed you how to leverage IIS's built-in authentication and encryption feature. You now know how to set up the IIS virtual root to allow certain authentication protocols and how to check a user's role membership in your components. I also showed you how to encrypt the HTTP traffic using SSL certificates.

You've also learned how you can use an additional component from Microsoft to secure and authenticate remoting traffic independently from the chosen transport format.

Last but not least you have had a close look at the new .NET Remoting security infrastructure included with the next version of the .NET Framework, which allows you to authenticate and/or impersonate the client's identity as well as securing traffic between the client and the server through digital signatures and encryption.

In the next chapter, you'll learn about some specialties of .NET Remoting. The chapter covers more advanced lifetime management issues, versioning, asynchronous calls, and events.

CHAPTER 6

Creating Remoting Clients

Now that you know how to create, deploy, and secure basic .NET Remoting components, this chapter will focus on creating different types of client applications. The reason for doing this is to show you how to configure .NET Remoting for each different type of client. Also, the hosting environment for the client has some implications on security—especially when it comes to authenticating against the server or accessing system resources.

Creating a Server for Your Clients

Before you start digging into the details for creating different types of .NET Remoting clients, you need a server that you can call. To keep things simple, your server will run in a console application itself, as you have already seen creation of different types of servers in Chapter 4.

For all your applications, you will use the same shared assembly named General where you put the server interfaces as well as the classes serialized across the network. For simplicity, also put the RemotingHelper, which you know from Chapter 4, into the shared assembly. The shared assembly defines two server interfaces: IRemoteFactory, which will be used for your primary server, and IRemoteSecond, which will be used later (for now it is not important). Also, a person class similar to the one of Chapter 5 is defined as you can see in Listing 6-1.

Listing 6-1. *The Shared Assembly for the Sample Applications*

```
using System;

namespace General
{
  public interface IRemoteFactory
  {
    Person GetPerson();
  }
```

```
  public interface IRemoteSecond
  {
    int GetNewAge();
  }

   [Serializable]
  public class Person
  {
    public int Age;
    public string Firstname, Lastname;

    public Person(string first, string last, int age)
    {
      this.Age = age;
      this.Firstname = first;
      this.Lastname = last;
    }
  }
}
```

Your primary server is a console application and just implements the IRemoteFactory interface and is configured in the application's configuration file. Listing 6-2 shows the implementation of the server.

Listing 6-2. *The Implementation of the Primary Server*

```
using System;
using System.Runtime.Remoting;

using General;

namespace Server
{
  public class ServerImpl : MarshalByRefObject, IRemoteFactory
  {
    private int _ageCount = 10;

    public Person GetPerson()
    {
      System.Console.WriteLine(">> Incoming request...");
      System.Console.WriteLine(">> Returning person {0}...", _ageCount);

      Person p = new Person("Test", "App", _ageCount++);
      return p;
    }
  }

  class ServerApp
  {
```

```
    [STAThread]
    static void Main(string[] args)
    {
      System.Console.WriteLine("Starting server...");
      RemotingConfiguration.Configure("Server.exe.config");

      System.Console.WriteLine("Server configured, waiting for requests!");
      System.Console.ReadLine();
    }
  }
}
```

The server will be configured with a TCP channel listening on port 1234 and runs as a server-activated Singleton object as you can see via the following configuration file:

```
<configuration>
  <system.runtime.remoting>
    <application>
      <channels>
        <channel ref="tcp" port="1234" />
      </channels>
      <service>
        <wellknown type="Server.ServerImpl, Server"
                   objectUri="MyServer.rem"
                   mode="Singleton" />
      </service>
    </application>
  </system.runtime.remoting>
</configuration>
```

For all the clients you will create in the following sections, you will use this server. Just for testing purposes, you start with creation of a console client to verify whether the server works or not. You should already know how to implement a console client, so I will go through the steps very briefly.

Creating a Console Client

The simplest .NET Remoting clients you can create are console applications. That was the reason for using them in the previous chapters. Configuration is put into the application configuration (Exename.exe.config) but can be put in any other file, too. In your console application project, which you can create with Visual Studio, just add references to System.Runtime.Remoting.dll and to your shared assembly General.dll. Afterwards, add a new item to your application—an *application configuration file*. In Listing 6-3, you can see the very simple implementation of the console client.

Listing 6-3. *Console Client Implementation*

```
using System;
using System.Runtime.Remoting;
using System.Runtime.Remoting.Activation;
```

```
using General;
using General.Client;

namespace ConsoleClient
{
  class ClientApp
  {
     [STAThread]
    static void Main(string[] args)
    {
      System.Console.WriteLine("Configuring client...");
      RemotingConfiguration.Configure("ConsoleClient.exe.config");

      System.Console.WriteLine("Calling server...");
      IRemoteFactory factory =
           (IRemoteFactory)RemotingHelper.CreateProxy(typeof(IRemoteFactory));
      Person p = factory.GetPerson();
      System.Console.WriteLine(">> Person retrieved: {0} {1}, {2}",
                                  p.Firstname, p.Lastname, p.Age.ToString());
      System.Console.WriteLine();
    }
  }
}
```

You can see the client configuration file in the following code excerpt. The client configures the TCP channel as well as the well-known server object running on localhost on port 1234.

```
<configuration>
  <system.runtime.remoting>
    <application>
      <channels>
        <channel ref="tcp" />
      </channels>
      <client>
        <wellknown type="General.IRemoteFactory, General"
                   url="tcp://localhost:1234/MyServer.rem" />
      </client>
    </application>
  </system.runtime.remoting>
</configuration>
```

In the client code just shown, the RemotingHelper class introduced in Chapter 4 is used for instantiating the client proxy class. Now that you have created the client, you can test the application. In Figures 6-1 and 6-2 you can see the client as well as the server.

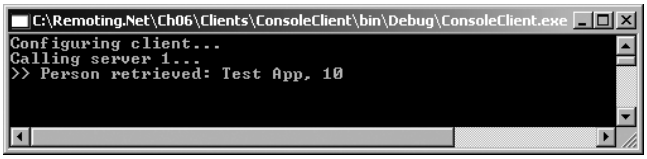

Figure 6-1. *The running client*

Figure 6-2. *The server in action*

But there is one issue with this easy kind of configuration that is important when configuring .NET Remoting clients in general. You cannot configure more than one server object in your client configuration with partially different settings for several reasons:

- First of all, you can have only one <application> element in your configuration file. Although you can register a channel of the same type multiple times with different settings (this means different formatters and providers) in your configuration, you cannot specify which channel to use with which <client> configuration part. Therefore, the channel with the highest priority will be taken by the runtime automatically. And if the settings of this channel do not fit with the server's channel, you'll run into an exception. The only way of solving this issue is manually configuring channels in your application.
- If your application wants to use two different servers providing objects based on the same interface, you cannot do so with one single client configuration because each type (e.g., the IRemoteFactory interface) can only be registered once. If you try registering a type more than once (for example, through the <wellknown /> configuration element), you'll get an exception with the following error message: "Attempt to redirect activation of type 'General.IRemoteFactory, General' which is already redirected."

In any of these cases, you either encapsulate communications with different servers in different application domains (remoting configuration works per AppDomain) or configure your channels and objects manually. If you decide to use a manual remoting configuration, you can use either a custom configuration file, the <appSettings> section within the application configuration, or your own configuration section (which means that you have to implement a configuration section handler) to avoid hard coding any server address parts in your code.

Imagine that you have the following configuration file for your client application that configures the server URLs through the app.config's <appSettings> configuration section:

```
<configuration>
  <appSettings>
```

```
    <add key="Server1" value="tcp://localhost:1234/MyServer.rem" />
    <add key="Server2"
         value="http://localhost/ClientWebRemoting/SecondIntermed.soap" />
  </appSettings>
</configuration>
```

In this case, you can use the following code for getting the URLs of the two servers from the configuration files when using manual configuration of .NET Remoting:

```
// calling first server
Console.WriteLine("\r\nCalling first server...");
url = ConfigurationSettings.AppSettings["Server1"];

ChannelServices.RegisterChannel(new TcpChannel());
factory = (IRemoteFactory)Activator.GetObject(
                        typeof(IRemoteFactory), url);

p = factory.GetPerson();
System.Console.WriteLine(">> Person retrieved: {0} {1}, {2}",
                     p.Firstname, p.Lastname, p.Age.ToString());
System.Console.WriteLine();

// calling second server
Console.WriteLine("Calling second server...");
url = ConfigurationSettings.AppSettings["Server2"];
ChannelServices.RegisterChannel(new HttpChannel());
factory = (IRemoteFactory)Activator.GetObject(
                        typeof(IRemoteFactory), url);

p = factory.GetPerson();
System.Console.WriteLine(">> Person retrieved: {0} {1}, {2}", p.Firstname,
p.Lastname, p.Age.ToString());
System.Console.WriteLine();
```

For client-activated objects, you have to use the `Activator.CreateInstance()` method instead of the `Activator.GetObject()` method. When doing so, you have to specify the type as well as an activation URL within the parameters as can be seen in the following code snippet:

```
string Url = ConfigurationSettings.AppSettings["Server1"];
System.Runtime.Remoting.Activation.UrlAttribute urlattr =
        new System.Runtime.Remoting.Activation.UrlAttribute (Url);

object[] actparams = {urlattr};
server = (YourServerType)Activator.CreateInstance(
                      typeof(YourServerType),null, actparams);
```

The UrlAttribute class can be found in the System.Runtime.Remoting.Activation namespace and specifies the activation URL for the client-activated object in that case.

Creating Windows Forms Clients

Creating a Windows Forms application that is a .NET Remoting client is nearly as simple as creating a console application client. First of all, just create a new Windows Forms project using Visual Studio, and then add the references to the shared assembly (General) as well as the .NET Remoting assemblies (System.Runtime.Remoting.dll) as you did before.

The first thing I always do when creating Windows Forms applications is to modify the project created with Visual Studio so that the Main function is in a different source file and not encapsulated directly into the form. This is a personal preference, but it allows me to more easily find the application's entry point. Therefore, I open the source file of the form created by Visual Studio and remove the `public static void Main()` method. Afterwards I create a new source file (usually called ApplicationName.cs) where I add the application general code. The content of this file is shown in Listing 6-4.

Listing 6-4. *The Application General Code*

```
using System;
using System.Windows.Forms;
using System.Runtime.Remoting;
using System.Runtime.Remoting.Activation;

using General;
using General.Client;

namespace WinClient
{
  public class WinApplication
  {
    private static IRemoteFactory _factory = null;

    public static IRemoteFactory ServerProxy
    {
      get
      {
        if(_factory == null)
        {
          _factory = (IRemoteFactory)RemotingHelper.CreateProxy(
                                typeof(IRemoteFactory));
        }

        return _factory;
      }
    }

    public static void Main(string[] args)
    {
      // first of all, configure remoting services
      RemotingConfiguration.Configure(
```

```
            AppDomain.CurrentDomain.SetupInformation.ConfigurationFile);
        RemotingConfiguration.Configure("WinClient.exe.config");

        // create the windows form and start message looping
        Application.Run(new MainForm());
      }
    }
}
```

The important thing you can see in the preceding code is the fact that I call `RemotingConfiguration.Configure()` at application startup. In general, it is unnecessary to call this method before each method call (and if you call it a second time afterwards, you will get an exception because channels and application URLs are already configured). Therefore, my configuration, which acts as a template for each proxy generation, is done exactly once.

The other thing I am using in the preceding code is a factory method for the proxy to the server. This way I can use one proxy for the server-activated object for the whole application without creating dozens of different proxies (if my application would use the proxy at several positions in code). This strategy can be used especially with SAOs of type single call because they are activated on the server, and a connection from the client to the server is established on a per-call basis (except you return MarshalByRefObjects).

In general, whether you can use this pattern for proxy creation or not depends on your situation. If you need to isolate communication of each form in your application with a .NET Remoting server, creating proxies at a per-form level might be more useful. I just wanted to show you another pattern that might be useful in some cases.

Now that you have created the application's main method, you can focus on creating a form that executes the calls to the server. The form I want to use consists of just two controls, a multiline text box named TextResults and a command button named ActionCall. Figure 6-3 shows the layout of the form (in the running application).

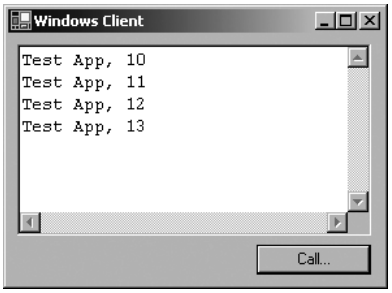

Figure 6-3. *The Windows application in action*

The .NET Remoting server will be called in the click event of the command button, as you can see in the following code snippet:

```
private void ActionCall_Click(object sender, System.EventArgs e)
{
```

```
  // get the transparent proxy for the factory
  IRemoteFactory proxy = WinApplication.ServerProxy;
  Person p = proxy.GetPerson();

  TextResults.AppendText(
    string.Format("{0} {1}, {2}\r\n", p.Firstname, p.Lastname, p.Age));
}
```

The server proxy is retrieved from the static `ServerProxy` property of the application class WinApplication introduced in Listing 6-4. Afterwards the application calls the server's method and appends the results to the text box of the form.

The rules for configuring .NET Remoting in Windows clients are exactly the same as for console application. The only difference that you have to keep in mind is to not configure .NET Remoting for each call—you can do that once at application startup, and the best way for doing so is in the application's main method.

■ **Note** If you keep state on the server and don't call the server for a while, you have to keep lifetime management in mind. In my example, the server keeps a counter state that is used for initializing the person's `age` property. If you wait a while (five minutes is the default), the state is lost because the .NET Remoting infrastructure releases the server's resources and automatically creates a new server instance, and, of course, any information stored in the previous instance of the server is lost. You can configure lifetime, too. For more information about lifetime management, take a look at Chapter 7.

Creating Back-End–Based Clients

The client applications you have seen so far are all "front-end" based. This means they are typically started by an end user and running within the end user's logon session. In the following two chapters, I want to go into details about creating clients that typically run on a server within the logon session of a service account. Examples of such clients are ASP.NET-based applications (which run either as ASPNET within the Web server process or a worker process if IIS 6.0 is used).[1]

Basically, configuration and calls work the same way with front-end clients as shown earlier. But when it comes to configuration in ASP.NET-based applications, as well as to security, there are some differences that I want to show you now.

ASP.NET-Based Clients

The first thing I want to show you is creating an ASP.NET-based application (either Web application or Web service) that calls the .NET Remoting server created at the very beginning of this chapter.

---

1. On Windows 2000 or Windows XP with IIS 5.x, any ASP.NET application usually runs in an external process called aspnet_wp.exe (ASP.NET worker process). The identity of the worker process can be configured in the processModel, too. On Windows Server 2003 with IIS 6.0, you can configure application pools. Each pool results in a separate process running under its own identity (default is NetworkService).

All you need to do is create an ASP.NET Web application project using Visual Studio .NET and add the references to your System.Runtime.Remoting.dll and General.dll assemblies as you did before. Your Web application will contain a Default.aspx page with just a list box and a button Web control as you can see in Figure 6-4.

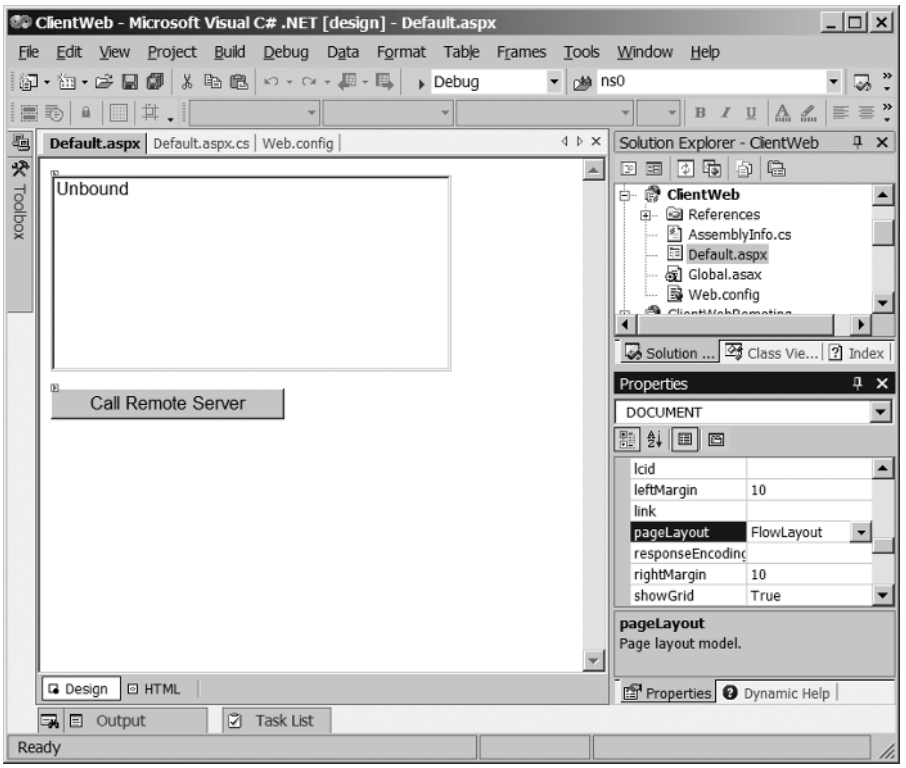

Figure 6-4. *Designer for your Web application client*

ASP.NET-based applications are usually configured through a web.config file that is automatically created when creating a new Web application project with Visual Studio. Generally, if you add .NET Remoting configuration to your web.config file, <service> tags are configured automatically while <client> configurations are ignored. For clarity, it is recommended to configure your .NET Remoting client configuration outside of web.config.

Just take a look at the following configuration, which you will use for configuring your ASP.NET client. I have included the configuration in web.config for the first step and will show you the problems with that afterwards.

```
<configuration>

  <system.runtime.remoting>
    <application>
      <channels>
        <channel ref="tcp" />
```

```
      </channels>
      <client>
        <wellknown type="General.IRemoteFactory, General"
                   url="tcp://localhost:1234/MyServer.rem" />
      </client>
    </application>
  </system.runtime.remoting>

  <system.web>
    <compilation defaultLanguage="c#" debug="true" />
    <customErrors mode="RemoteOnly" />
    <authentication mode="Windows" />
    <authorization>
      <allow users="*" />
    </authorization>
    <sessionState mode="InProc"
                  stateConnectionString="tcpip=127.0.0.1:42424"
                  sqlConnectionString=
                      "data source=127.0.0.1;Trusted_Connection=yes"
                  cookieless="false" timeout="20" />
    <globalization requestEncoding="utf-8" responseEncoding="utf-8" />
  </system.web>

</configuration>
```

But why configure the application now? The naïve way would simply configure remoting in the Page_Load or in the event procedure of the button where the server component will be called. But if you do this, you will fail with the second request—but why?

Actually, the reason is very simple. As mentioned before, `RemotingConfiguration.Configure()` configures .NET Remoting on an AppDomain basis. This means within the application domain channels and remote objects are registered. Of course, each channel must have a unique name, and each type can be configured only once within one AppDomain.

In ASP.NET, for each virtual root in IIS an instance of HttpApplication is launched by the ASP.NET runtime. Each HttpApplication on its own resides in a separate application domain. The application itself exists as long as the host process[2] is alive—this means the application stays the same between different browser requests. If you configure .NET Remoting in one of the page-level events, you try to configure the same channels and well-known objects multiple times for the application domain of the HttpApplication (which stays alive between the requests).

The solution again is fairly simple. ASP.NET has the capability to catch application-wide events within the Global.asax, which is created automatically when creating an ASP.NET project with Visual Studio. The Global.asax actually is a class for your own Web application that inherits from System.Web.HttpApplication. The important event in this case is the Application_Start event of the application. The following code fragment shows how to configure your application within this event, which you can find in the code-behind for the Global.asax file (Global.asax.cs):

2. As you have seen before, the host process can be the Web server itself, the ASP.NET worker process aspnet_wp.exe, or on machines with an IIS 6.0 worker process for the application pool.

```
protected void Application_Start(Object sender, EventArgs e)
{
  // configure the remoting server
  RemotingConfiguration.Configure(Server.MapPath("web.config"));
}
```

The path to the web.config file is retrieved through `Server.MapPath()`, which returns the physical path to the file on the local system. The call to your server component is again fairly simple. The following code snippet shows the event handler for the command button you have seen previously in Figure 6-4:

```
private void ActionCall_Click(object sender, System.EventArgs e)
{
  IRemoteFactory proxy = (IRemoteFactory)RemotingHelper.CreateProxy(
                                          typeof(IRemoteFactory));
  Person p = proxy.GetPerson();

  ListResults.Items.Add(string.Format("{0} {1}, {2}",
                              p.Firstname, p.Lastname, p.Age));
}
```

That's it, now you have created an ASP.NET-based client for a .NET Remoting server component. Most importantly, remember to configure .NET Remoting within the Application_Start event of the Global.asax code-behind file.

Remoting Components Hosted in IIS As Clients

As you already know from the previous chapters, it is possible to host .NET Remoting components in IIS, too. In this case, the ASP.NET runtime itself hosts the server component, and configuration of the server is done through web.config.

Any <service> configuration found in web.config is automatically handled by ASP.NET, which means that you don't need to call `RemotingConfiguration.Configure()` for your server configuration. If a client configuration is found in web.config, it will be ignored by the ASP.NET runtime—therefore clients must be configured manually.

Although configuring clients in web.config is basically possible, I will not recommend doing so for reasons discussed in the last chapter. It is better to add another configuration file to your solution (even if you are writing simple ASP.NET Web applications or Web servers that don't host remoting components) and do your client configuration in this file.

So you can see how to configure a .NET Remoting component hosted in IIS as a client for another server (the server you created at the very beginning of the chapter), start with creating a Web application project with Visual Studio that you will use for your .NET Remoting server component. Because you don't need it, you can delete the ASPX page that is created automatically after Visual Studio has created the project. The other files, especially web.config and Global.asax, will be necessary for your component.

After you have added the references to System.Runtime.Remoting as well as General.dll, you can add a class to your project that will be the server. The server's implementation is shown in Listing 6-5.

Listing 6-5. *An Intermediary .NET Remoting Server Hosted in IIS*

```
using System;
using System.Runtime.Remoting;

using General;
using General.Client;

namespace ClientWebRemoting
{
  public class SecondServer : MarshalByRefObject, IRemoteSecond
  {
    private int _counter = 1;
    private IRemoteFactory _proxy;

    public SecondServer()
    {
      System.Diagnostics.Debug.WriteLine("Initializing server...");

      _proxy = (IRemoteFactory)RemotingHelper.CreateProxy(
                                          typeof(IRemoteFactory));

      System.Diagnostics.Debug.WriteLine("Server initialized!");
    }

    public int GetNewAge()
    {
      Person p = _proxy.GetPerson();
      int ret = p.Age + (_counter++);

      System.Diagnostics.Debug.WriteLine(
                ">> Incoming request returns " + ret.ToString());

      return ret;
    }
  }
}
```

Remember that in the preceding code you don't use any client-side remoting configuration because you are hosting the component in ASP.NET and IIS. Therefore, you have to do the same thing as you did before—perform the client configuration within the Global.asax Application_Start event. I'll show the configuration for this component as client of the other server later in this chapter.

Now you know why you need to define the interface IRemoteSecond in the shared assembly at the very beginning of this chapter—your second server is implementing exactly this interface now.

■ **Note** Because you don't have a console window for viewing what happens on the server, use `Debug.WriteLine()` so that you can view activity in the output window from Visual Studio when debugging the solution.

Because the component is hosted in IIS, you can use the HTTP channel only. Therefore, the configuration of the server component in the web.config file looks like the following:

```
<configuration>

  <system.runtime.remoting>
    <application>
      <channels>
        <channel ref="http" />
      </channels>
      <service>
        <wellknown type="ClientWebRemoting.SecondServer, ClientWebRemoting"
                   objectUri="SecondServer.soap"
                   mode="Singleton" />
      </service>
    </application>
  </system.runtime.remoting>

  <system.web>
    <compilation defaultLanguage="c#" debug="true" />
    <authentication mode="None" />
  </system.web>
</configuration>
```

The service configuration is done automatically by the ASP.NET runtime. You want to call your other remoting server component from within this server, so you need to add the client configuration. Adding the client configuration to the web.config would be a very bad idea in this case. Remember that `RemotingConfiguration.Configure()` configures anything in the <system.runtime.remoting> section. If you just add your client's configuration and try to call `RemotingConfiguration.Configure()`, an exception will occur because the .NET Remoting runtime tries to configure the client and the server. Remember that the server has already been configured by the ASP.NET runtime. Therefore, in this situation you would register the server as well as the server's channel twice, which is not possible within an application domain.

A simple, effective, and in my opinion much more readable solution is to add another configuration file (e.g., RemotingClient.*config*) to the project and add the client configuration to this file. The client configuration is shown in the following code snippet:

```
<configuration>
  <system.runtime.remoting>
    <application>
      <channels>
        <channel ref="tcp" />
```

```
      </channels>
      <client>
        <wellknown type="General.IRemoteFactory, General"
                   url="tcp://localhost:1234/MyServer.rem" />
      </client>
    </application>
  </system.runtime.remoting>
</configuration>
```

■ **Note** I'd also recommend creating your own client configuration files for ASP.NET Web applications and Web services.

As you did before in the ASP.NET client application, you perform client configuration within the Application_StartUp event of the Global.asax code behind as you can see in the following code snippet:

```
protected void Application_Start(Object sender, EventArgs e)
{
     RemotingConfiguration.Configure(Server.MapPath("RemotingClient.config"));
}
```

This time you are using your own configuration file RemotingClient.config, which is placed in the application's root directory.

■ **Note** The file that contains the configuration can be any file; the file extension doesn't matter. Nevertheless, it is a good idea to use files with a .config extension because they are protected by the ASP.NET runtime by default and not returned to the client, while other files with extensions like .txt are by default not protected by the ASP.NET runtime and therefore can be browsed using the browser when the URL is known.

Because you have created a Web application project, you can debug without manually attaching to the ASP.NET worker process or the IIS 6.0 worker process. But Visual Studio wants you to select a start page for debugging. Now you have two possibilities: either add a dummy page that is used as a start page for debugging or just select the Global.asax or another file as the start page. In this case, you will see an error in the browser, but debugging works anyway.

For testing your intermediary .NET Remoting server, you will need to create another client that calls this server through the IRemoteSecond interface. For simplicity, create a console client. Take a look at the client's implementation in Listing 6-6.

Listing 6-6. *A Client for Your Second Server Component*

```
using System;
using System.Runtime.Remoting;
```

```
using General;
using General.Client;

namespace ClientOfWebRemoting
{
  class ClientApp
  {
     [STAThread]
    static void Main(string[] args)
    {
      System.Console.WriteLine("Configuring client...");
      RemotingConfiguration.Configure("ClientOfWebRemoting.exe.config");

      System.Console.WriteLine("Calling server...");
      IRemoteSecond second = (IRemoteSecond)RemotingHelper.CreateProxy(
                                                    typeof(IRemoteSecond));
      for(int i=0; i < 5; i++)
      {
        System.Console.WriteLine("Result: {0}", second.GetNewAge());
      }

      System.Console.ReadLine();
    }
  }
}
```

The client itself, of course, is configured with the HTTP channel. This time you don't have to specify any port information because the server is hosted in IIS and can be reached at port 80.

```
<configuration>
  <system.runtime.remoting>
    <application>
      <channels>
        <channel ref="http" />
      </channels>
      <client>
        <wellknown type="General.IRemoteSecond, General"
                   url="http://localhost/ClientWebRemoting/SecondServer.soap" />
      </client>
    </application>
  </system.runtime.remoting>
</configuration>
```

Now that you have implemented the intermediary service and a client for the service, you can test the application. Just start debugging the .NET Remoting component and the client (for server activity take a look at the Visual Studio output window because you are using `Debug.WriteLine()` for logging activity). Figures 6-5, 6-6, and 6-7 demonstrate the activities in the client, the intermediary server, and the final server, respectively.

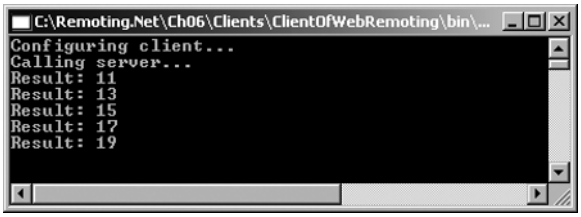

Figure 6-5. *The end-user client for the IIS hosted component*

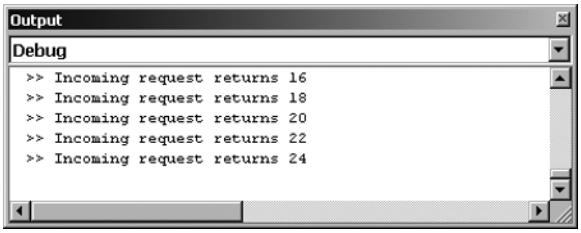

Figure 6-6. *The output window for the IIS hosted server*

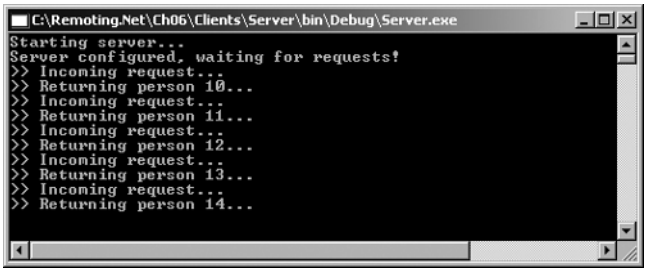

Figure 6-7. *The console window for your final server*

Security Considerations

The last thing I want to discuss is the difference of security between end-user clients and back-end–based clients. Primarily, you have to keep in mind that end-user clients are running under the end-user's identity, whereas back-end–based clients run under a service account.

If the .NET Remoting server from Chapter 1 requires authentication, the primary question is which users are accepted. In the case of end users, you need to flow the identity from your client front end to your server back end through each intermediary. That's not a big deal if all runs on one machine, but I think you agree with me that this is not the usual situation.

In most cases, the client, intermediary servers, and back-end servers are all running on different machines. To flow the end user's identity from the very front end to the very back end, you need Kerberos. Furthermore, Kerberos must be configured properly. Remember the concept of delegation I mentioned in Chapter 5? When using Windows-based systems, delegation is

a separate privilege. To enable delegation, you must specifically allow machines to delegate tickets they get from the client. That's a configuration option that must be set at Active Directory domain and machine level.

On the other hand, flowing the identity from the very front end to the very back end is something that leads to extensive management tasks and potential risks as you now have to trust all your end users to adhere to sensible password management practices. As soon as the very back end accesses resources and impersonates the client (that's necessary for the intermediary to pass the token to the back end), you have to configure all access control lists on the intermediary as well as the back end properly. Figure 6-8 demonstrates the situation described earlier.

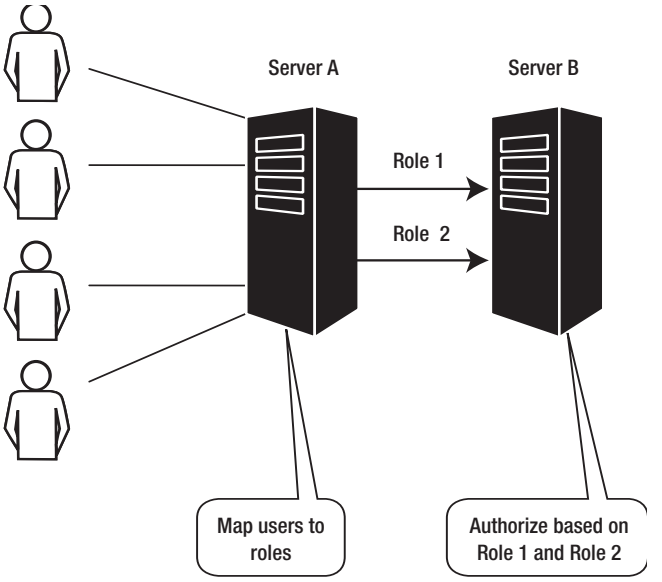

Figure 6-8. *Impersonation, flowing the user to the back end*

And it's exactly the intense configuration overhead that makes *impersonation and delegation fragile and dangerous*. Any misconfiguration can lead to a potential security hole (although the application itself might be secure—configuration and applications have to work together for really secure systems). Therefore, it should only be used if it is inherently necessary.

A better way would be using roles for logically related groups of users and letting the service impersonate a specific user for each role. Doing so makes management much easier because you don't have to think about hundreds of users but only about a couple for roles you have to manage. This strategy is called *trusted subsystem*. The concept is shown in Figure 6-9.

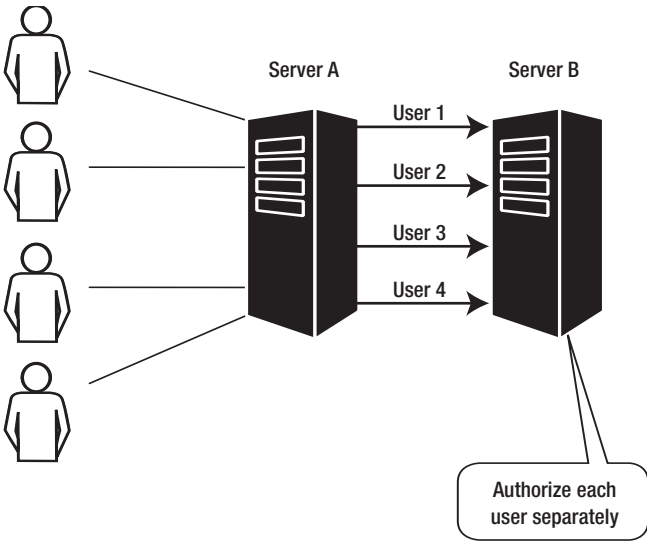

Figure 6-9. *Trusted subsystem*

If you don't impersonate specific users (or users for roles as you would do in the trusted subsystem), on the other hand, you have to keep in mind that service accounts are often just accounts of the local machine. Therefore, authentication of a service account on another machine will fail (the ASPNET account on machine A is not the same as the ASPNET account on machine B). In this case, you have two possibilities: either create domain accounts for the services or flow identity through impersonation (but do that based on roles, not for each user!).

■ **Caution** If you are configuring service accounts at a domain level, keep the principle of *least privilege* in mind. Never configure too many privileges for such accounts to limit damage if one of your services gets hacked. Configure as few privileges as possible.

The last option for flowing identities across machines would be manually implementing your own protocols. But never underestimate the effort to build such protocols that are as secure as, for example, Kerberos is.

Now that you have seen the concepts you have to keep in mind, just take a short look at how security behaves by configuring the back-end server using the SSPI security solution demonstrated in the last chapter.

Let's start with the simplest example, the Windows Forms client created in the previous chapter calling the .NET Remoting server directly with SSPI configured. First of all, you have to add references to the Microsoft.Samples.Security.SSPI.dll and Microsoft.Samples.Remoting.Security.dll assemblies as explained in Chapter 5.

Next, change the server to output the security information sent from the client to the server. For this purpose, you will modify the server's configuration as you can see in the following code snippet:

```
<configuration>
  <system.runtime.remoting>
    <application>
      <channels>
        <channel ref="tcp" port="1234">
          <serverProviders>
            <provider type="Microsoft.Samples.Runtime.Remoting.
                            Security.SecurityServerChannelSinkProvider,
                            Microsoft.Samples.Runtime.Remoting.Security"
                      securityPackage="negotiate"
                      authenticationLevel="packetPrivacy" />
            <formatter ref="binary" />
          </serverProviders>
        </channel>
      </channels>
      <service>
        <wellknown type="Server.ServerImpl, Server"
                   objectUri="MyServer.rem"
                   mode="Singleton" />
      </service>
    </application>
  </system.runtime.remoting>
</configuration>
```

Next, you change the server's code to output the authenticated user after being called. The following code snippet shows the changed `GetPerson()` method of the server:

```
public Person GetPerson()
{
  System.Console.WriteLine(">> Incoming request...");
  System.Console.WriteLine(">> Returning person {0}...", _ageCount);

  IPrincipal user = System.Threading.Thread.CurrentPrincipal;
  if(user != null)
  {
    System.Console.WriteLine(">> >> Authenticated user: {0}, {1}",
                  user.Identity.Name, user.Identity.AuthenticationType);
  }
  else
  {
    System.Console.WriteLine(">> >> Unauthenticated user!!");
  }
```

```
  Person p = new Person("Test", "App", _ageCount++);
  return p;
}
```

With the next step you change the configuration of the client to enable the .NET Remoting security solution. In the client, you have to change the configuration only. The following code snippet shows how to change the channel configuration in the Windows Forms client's application configuration file:

```
<channel ref="tcp">
  <clientProviders>
    <formatter ref="binary" />
    <provider type="Microsoft.Samples.Runtime.Remoting.
                    Security.SecurityClientChannelSinkProvider,
                    Microsoft.Samples.Runtime.Remoting.Security"
              securityPackage="negotiate"
              impersonationLevel="identify"
              authenticationLevel="packetPrivacy" />
  </clientProviders>
</channel>
```

■ **Note** In both configuration files, the client's and the server's file, the type attribute of the <provider> tag may not have line breaks. I have added them just for readability.

Not very surprisingly, the output of the server looks like what appears in Figure 6-10 after you have called the server from the client started with the test user account.

Figure 6-10. *The server called from a simple client*

Now modify the ASP.NET application client for your .NET Remoting server by just changing the Remoting client configuration as you did with the Windows Forms version of the client, only add the SSPI provider as you can see in the preceding code snippet. Run the client and take a look at the server's output (don't forget to add the reference to the Microsoft sample assemblies as you did before), which should resemble what you see in Figure 6-11.

Figure 6-11. *The server called from the ASP.NET client application*

As you can see, now the server has authenticated the ASP.NET account. Remember that this would not work across machine borders because the ASP.NET account of machine A is not known on machine B. The solution for this would be to configure a domain account (with least privileges) and configure to run ASP.NET under this account (either through <processModel> in machine.config on IIS 5.x or through the application pool identity on IIS 6.0).

The last option I want to show you is the case when the ASP.NET application impersonates the client. For this reason, add the following tag to your web.config file of your ASP.NET client application:

```
<identity
    impersonate="true" />
```

In Figure 6-12, you can see how this affects your authentication. In this case, the identity flows from the browser to the Web server and from the Web server to the .NET Remoting server component.

Figure 6-12. *Identity flows from the client through the Web server to the remoting server.*

But when you take a look at Figure 6-12, you can see that the user seems to be definitely not the real end user. When you take a closer look at the web.config file of the ASP.NET client, you see that it is configured with Windows authentication. But what is wrong with the configuration now? Well, the solution can be found in the IIS configuration where you have enabled anonymous access, as you can see in Figure 6-13.

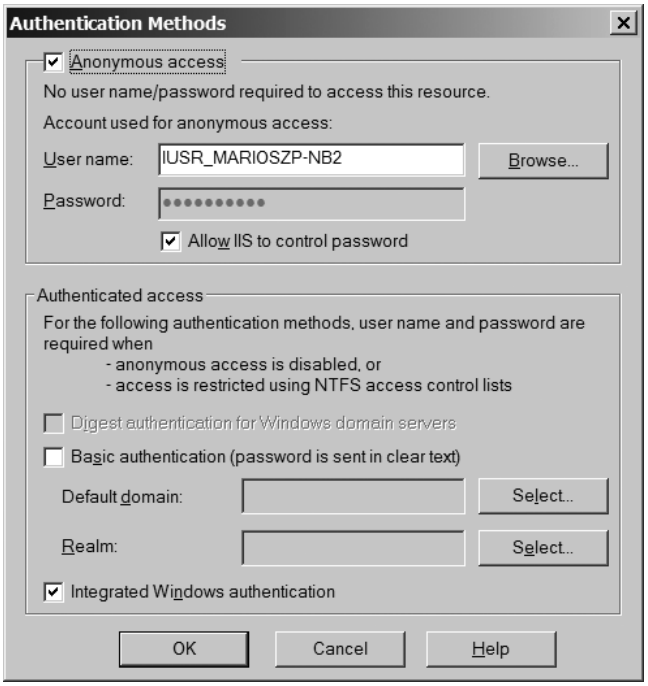

Figure 6-13. *IIS configuration of your application*

If you change the configuration to just enable Windows authentication, the result will be quite different. Now let's take a last look at the output of the server after you have disabled anonymous authentication in IIS and started a browser with your test user account (see Figure 6-14).

Figure 6-14. *Server output after IIS anonymous authentication is disabled*

■ **Note** When creating ASP.NET projects using Visual Studio .NET, anonymous access is automatically enabled by Visual Studio. If you share a directory as a virtual directory using Windows Explorer, anonymous access is by default enabled on Windows XP SP1 but disabled on Windows XP SP2 or Windows Server 2003 by default.

Don't forget that delegation must be enabled if you want to flow the identity of the end user through the Web server to the back-end remoting server across machine boundaries. For the purposes of this example, I tried all the samples running on the same machine for simplicity.

Summary

In this chapter, you took a look into the details for creating different types of .NET Remoting clients. You saw the different ways for configuring clients and also saw that you have to be careful when configuring .NET Remoting in ASP.NET-based client applications.

In the last section of this chapter, you took a closer look at some security concerns—especially how you can flow the identity from the client to the server. In general, a very good strategy for flowing identities is having separate accounts for roles of users to not flow all the end users through all tiers of your distributed applications for easier management. Don't forget that Kerberos must be configured properly when identities flow across machine borders.

CHAPTER 7

In-Depth .NET Remoting

As you've already seen in the previous chapters, developers of distributed applications using .NET Remoting have to consider several fundamental differences from other remoting techniques and, of course, from the development of local applications. One of the major issues you face in any distributed object framework is the decision of how to manage an object's lifetime. Generally you have two possibilities: using distributed reference counting/garbage collection or using time-to-live counters associated with each object.

Managing an Object's Lifetime

Both CORBA and DCOM have employed distributed reference counting. With DCOM, for example, the server's objects keep counters of referrers that rely on the `AddRef()` and `Release()` methods in the same way as it's done for common COM objects. Unfortunately this has some serious drawbacks: each call to increase or decrease the reference counter has to travel to the remote object without any "real" application benefit (that is, the remote call does not do any "real" work). In DCOM, the clients will also ping the server at certain intervals to signal that they are still alive.

Both pinging and the calls to change the reference counter result in an increased network load, and the former will very likely not work with some firewalls or proxies that only allow stateless HTTP connections to pass through.

Because of those implications, Java RMI introduced a lease-based lifetime service that bears a close resemblance to what you can see in .NET Remoting today. The lease-based concept essentially assigns a time-to-live (TTL) count to each object that's created at the server. A LeaseManager then polls all server-side objects at certain intervals and decrements this TTL. As soon as this time reaches zero, the object is marked as timed out and will be marked for garbage collection. Additionally, for each method call placed on the remote object, the TTL is incremented again to ensure that objects currently in use will not time out.

In reality, though, there are applications in which objects may exist that are not used all the time. A pure TTL-based approach would time-out these objects too soon. Because of this, the .NET Remoting framework also supports a concept called *sponsorship*. For each object, one or more sponsors might be registered. Upon reaching zero TTL, the LeaseManager contacts each sponsor and asks if it wants to increase the object's lifetime. Only when none of them responds positively in a given time is the object marked for garbage collection.

A sponsor itself is a MarshalByRefObject as well. It can therefore be located on the client, the server, or any other machine that is reachable via .NET Remoting.

Understanding Leases

A *lease* holds the time-to-live information for a given object. It is therefore directly associated with a certain MarshalByRefObject's instance. At the creation of a lease, the following information is set (all of the following are of type TimeSpan):

Property	Default	Description
`InitialLeaseTime`	5 minutes	The initial TTL after an object's creation.
`RenewOnCallTime`	2 minutes	The grace time for a method call that is placed on the object. Mind, though, that these times are not additive—for instance, calling a method a thousand times will not result in a TTL of 2,000 minutes, but one of 2 minutes.
`SponsorShipTimeout`	2 minutes	When sponsors are registered for this lease, they will be contacted upon expiration of the TTL. They then can contact the LeaseManager to request additional lease time for the sponsored object. When no sponsor reacts during the time defined by this property, the lease will expire and the object will be garbage collected.

Both the ILease interface and the Lease class that provides the standard implementation are located in System.Runtime.Remoting.Lifetime. Whenever a MarshalByRefObject is instantiated (either as a CAO or as a SAO—even when using Singleton mode), the framework calls the object's `InitializeLifetimeService()` method, which will return an ILease object. In the default implementation (that is, when you don't override this method), the framework calls `LifetimeServices.GetLeaseInitial()`, which returns a Lease object containing the defaults shown in the preceding table.

■ **Tip** Whenever I mention some class for which you don't know the containing namespace, you can use WinCV.exe, which is in the Framework SDK, to locate the class and get some information about its public interface. An even better (free) tool is .NET Reflector, available for download at `http://www.aisto.com/roeder/dotnet`.

The Role of the LeaseManager

The LeaseManager runs in the background of each server-side application and checks all remoted objects for their TTL. It uses a timer and a delegate that calls its `LeaseTimeAnalyzer()` method at certain intervals.

The initial value for this interval is set to 10 seconds. You can change this interval by using either the following line of code:

```
LifetimeServices.LeaseManagerPollTime = TimeSpan.FromSeconds(1);
```

or, when using configuration files, you can add the following settings to it:

```
<configuration>
   <system.runtime.remoting>
      <application>
         <lifetime leaseManagerPollTime="1s" />
```

```
      </application>
   </system.runtime.remoting>
</configuration>
```

You may specify different time units for the leaseManagerPollTime attribute. Valid units are D for Days, H for hours, M for minutes, S for seconds, and MS for milliseconds. When nothing is specified, the system will default to S; combinations such as "1H5M" are *not* supported.

Changing the Default Lease Time

You can easily change the default TTL for all objects in a given server-side app-domain in two ways. First, you can use the following code fragment to alter the application-wide initial lease times:

```
LifetimeServices.LeaseTime = System.TimeSpan.FromMinutes(10);
LifetimeServices.RenewOnCallTime = System.TimeSpan.FromMinutes(5);
```

As the preferred alternative, you can add the following sections to your configuration files:

```
<configuration>
   <system.runtime.remoting>
      <application>
         <lifetime
            leaseTimeout="10M"
            renewOnCallTime="5M"
         />
      </application>
   </system.runtime.remoting>
</configuration>
```

However, you have to be aware of the fact that this change affects each and every remote object that is published by the server application which uses this configuration file. Increasing the TTL therefore can have negative effects toward the memory and resource utilization of your application, whereas decreasing it can lead to objects being prematurely destroyed.

■ **Caution** Whenever a client places a method call to a remote object with an expired TTL, an exception will be thrown.

Changing the Lease Time on a Per-Class Basis

For certain MarshalByRefObjects (especially Singleton-mode services or objects published by `RemotingServices.Marshal()`), it is desirable to have either an "unlimited" TTL or a different lease time from that of other objects on the same server.

You can implement this functionality by overriding MarshalByRefObject's `InitializeLifetimeService()`. This method is defined to return an object, but later uses in the framework will cast this object to Lease, so make sure not to return anything else. For example, to provide a Singleton with unlimited lifetime, implement the following:

```
class InifinitelyLivingSingleton: MarshalByRefObject
{
   public override object InitializeLifetimeService()
   {
      return null;
   }
    // ...
}
```

To set a custom lifetime different from "infinity," you can call `base.InitializeLifetimeService()` to acquire the reference to the standard ILease object and set the corresponding values afterwards.

```
class LongerLivingSingleton: MarshalByRefObject
{

   public override object InitializeLifetimeService()
   {
      ILease tmp = (ILease) base.InitializeLifetimeService();
      if (tmp.CurrentState == LeaseState.Initial)
      {
         tmp.InitialLeaseTime = TimeSpan.FromSeconds(5);
         tmp.RenewOnCallTime = TimeSpan.FromSeconds(1);
      }
      return tmp;
   }
}
```

Examining a Basic Lifetime Example

In the following example, I show you how to implement the different changes in an object's lifetime in one application. The server will therefore export three MarshalByRefObjects as Singletons: DefaultLifeTimeSingleton, which will use the "base" lifetime set by a configuration file; LongerLivingSingleton, which will override `InitializeLifetimeService()` to return a different lease time; and finally InfinitelyLivingSingleton, which will just return null from `InitializeLifetimeServices()`.

As you can see in the following configuration file, I change the default lifetime to a considerably lower value so that you can observe the effects without having to wait five minutes until the objects time out:

```
<configuration>
  <system.runtime.remoting>
    <application>

      <channels>
        <channel ref="http" port="1234" />
      </channels>
```

```
      <lifetime
         leaseTime="10MS"
         renewOnCallTime="10MS"
         leaseManagerPollTime = "5MS"
      />

      <service>

        <wellknown mode="Singleton"
                   type="Server.DefaultLifeTimeSingleton, Server"
                   objectUri="DefaultLifeTimeSingleton.soap" />

        <wellknown mode="Singleton"
                   type="Server.LongerLivingSingleton, Server"
                   objectUri="LongerLivingSingleton.soap" />

        <wellknown mode="Singleton"
                   type="Server.InfinitelyLivingSingleton, Server"
                   objectUri="InfinitelyLivingSingleton.soap" />

      </service>
    </application>
  </system.runtime.remoting>
</configuration>
```

In the server-side implementation shown in Listing 7-1, I just include some `Console.WriteLine()` statements so that you can see when new objects are created by the framework. These components can be hosted by a simple .NET EXE that calls `RemotingConfiguration.Configure()` to read the preceding configuration file.

Listing 7-1. *Implementation Showing the Effects of Different Lifetime Settings*

```
using System;
using System.Runtime.Remoting.Lifetime;
using System.Runtime.Remoting;

namespace Server
{
   class DefaultLifeTimeSingleton: MarshalByRefObject
   {

      public DefaultLifeTimeSingleton()
      {
         Console.WriteLine("DefaultLifeTimeSingleton.CTOR called");
      }

      public void DoSomething()
      {
```

```
            Console.WriteLine("DefaultLifeTimeSingleton.DoSomething called");
        }
    }

    class LongerLivingSingleton: MarshalByRefObject
    {

        public override object InitializeLifetimeService()
        {
            ILease tmp = (ILease) base.InitializeLifetimeService();
            if (tmp.CurrentState == LeaseState.Initial)
            {
                tmp.InitialLeaseTime = TimeSpan.FromSeconds(5);
                tmp.RenewOnCallTime = TimeSpan.FromSeconds(1);
            }
            return tmp;
        }

        public LongerLivingSingleton()
         {
            Console.WriteLine("LongerLivingSingleton.CTOR called");
        }

        public void DoSomething()
        {
            Console.WriteLine("LongerLivingSingleton.DoSomething called");
        }
    }

    class InfinitelyLivingSingleton: MarshalByRefObject
    {
        public override object InitializeLifetimeService()
        {
            return null;
        }
        public InfinitelyLivingSingleton()
        {
            Console.WriteLine("InfinitelyLivingSingleton.CTOR called");
        }

        public void DoSomething()
        {
            Console.WriteLine("InfinitelyLivingSingleton.DoSomething called");
        }
    }

}
```

To develop the client, I simply referenced the server-side implementation DLL from the client.[1] In the example shown in Listing 7-2, the different Singletons will be called several times with varying delays.

Listing 7-2. *The Client Calling the Various SAOs with Different Delays*

```
using System;
using System.Runtime.Remoting;
using System.Threading;
using Server;
namespace Client
{
   class Client
   {
      static void Main(string[] args)
      {
         String filename = "client.exe.config";
         RemotingConfiguration.Configure(filename);

         DefaultLifeTimeSingleton def = new DefaultLifeTimeSingleton();
         LongerLivingSingleton lng = new LongerLivingSingleton();
         InfinitelyLivingSingleton inf = new InfinitelyLivingSingleton();

         /*** FIRST BLOCK ***/
         Console.WriteLine("Calling DefaultLifeTimeSingleton");
         def.DoSomething();
         Console.WriteLine("Sleeping 100 msecs");
         Thread.Sleep(100);
         Console.WriteLine("Calling DefaultLifeTimeSingleton (will be new)");
         def.DoSomething(); // this will be a new instance


         /*** SECOND BLOCK ***/
         Console.WriteLine("Calling LongerLivingSingleton");
         lng.DoSomething();
         Console.WriteLine("Sleeping 100 msecs");
         Thread.Sleep(100);
         Console.WriteLine("Calling LongerLivingSingleton (will be same)");
         lng.DoSomething(); // this will be the same instance
         Console.WriteLine("Sleeping 6 seconds");
         Thread.Sleep(6000);
         Console.WriteLine("Calling LongerLivingSingleton (will be new)");
         lng.DoSomething(); // this will be a new same instance
```

1. You could also use `soapsuds -ia:server -nowp -oa:generated_meta.dll` to generate a metadata assembly that will be referenced by the client application, but this is not recommended.

```
            /*** THIRD BLOCK ***/
            Console.WriteLine("Calling InfinitelyLivingSingleton");
            inf.DoSomething();
            Console.WriteLine("Sleeping 100 msecs");
            Thread.Sleep(100);
            Console.WriteLine("Calling InfinitelyLivingSingleton (will be same)");
            inf.DoSomething(); // this will be the same instance
            Console.WriteLine("Sleeping 6 seconds");
            Thread.Sleep(6000);
            Console.WriteLine("Calling InfinitelyLivingSingleton (will be same)");
            inf.DoSomething(); // this will be a new same instance

            Console.ReadLine();
        }
    }
}
```

In the first block, the client calls DefaultLifetimeSingleton twice. As the delay between both calls is 100 milliseconds and the object's lifetime is 10 milliseconds, a new instance of the SAO will be created.

The second block calls LongerLivingSingleton three times. Because of the increased lifetime of five seconds, the first two calls will be handled by the same instance. A new object will be created for the third call, which takes place after a six-second delay.

In the last block, the client executes methods on InfinitelyLivingSingleton. Regardless of which delay is used here, the client will always talk to the same instance due to the fact that `InitializeLifetimeService()` returns null, which provides infinite TTL. Figures 7-1 and 7-2 prove these points.

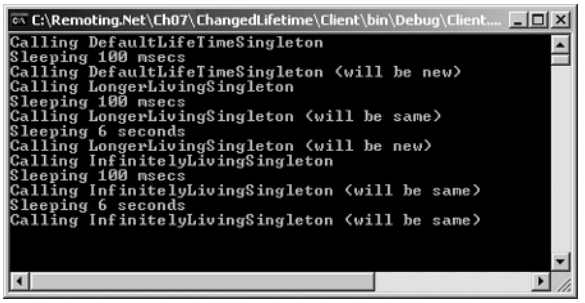

Figure 7-1. *Client-side output when dealing with different lifetimes*

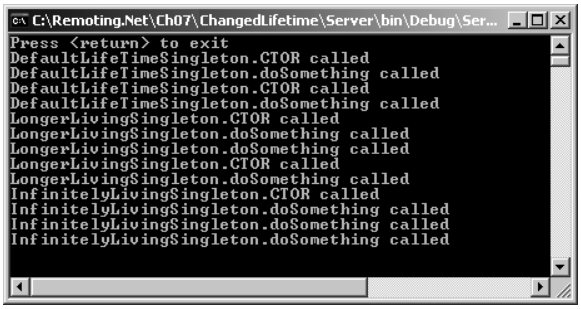

Figure 7-2. *Server-side output when dealing with different lifetimes*

Extending the Sample

The .NET Remoting framework only allows the default lifetime to be changed for all objects, which might invite you to hard code changes for class-specific TTLs. The problem here is that you might not necessarily know about each possible deployment scenario when developing your server-side components, so nondefault lifetimes should in reality be customizable by configuration files as well.

You can therefore change your applications to not directly derive from MarshalByRefObjects, but instead from an enhanced subtype that will check the application's configuration file to read and set changed lifetime values.

■ **Tip** I think it's good practice to use an extended form of MarshalByRefObject for your applications, as you might not always know which kind of common functionality you'll want to implement later.

The ExtendedMBRObject will override `InitializeLifetimeService()` and check the appSetting entries in the configuration file for nondefault lifetime information on a class-by-class basis. It is shown in Listing 7-3.

Listing 7-3. *Base Class for the Following Examples*

```
using System;
using System.Configuration;
using System.Runtime.Remoting.Lifetime;

namespace Server
{

   public class ExtendedMBRObject: MarshalByRefObject
   {
      public override object InitializeLifetimeService()
      {
         String myName = this.GetType().FullName;
```

```
            String lifetime =
               ConfigurationSettings.AppSettings[myName + "_Lifetime"];

            String renewoncall =
               ConfigurationSettings.AppSettings[myName + "_RenewOnCallTime"];

            String sponsorshiptimeout =
               ConfigurationSettings.AppSettings[myName + "_SponsorShipTimeout"];

            if (lifetime == "infinity")
            {
               return null;
            }
            else
            {
               ILease tmp = (ILease) base.InitializeLifetimeService();
               if (tmp.CurrentState == LeaseState.Initial)
               {
                  if (lifetime != null)
                  {
                     tmp.InitialLeaseTime =
                        TimeSpan.FromMilliseconds(Double.Parse(lifetime));
                  }

                  if (renewoncall != null)
                  {
                     tmp.RenewOnCallTime =
                        TimeSpan.FromMilliseconds(Double.Parse(renewoncall));
                  }

                  if (sponsorshiptimeout != null)
                  {
                     tmp.SponsorshipTimeout =
                      TimeSpan.FromMilliseconds(Double.Parse(sponsorshiptimeout));
                  }
               }
               return tmp;
            }
         }
      }
   }
}
```

In the following example, all server-side objects are changed to inherit from ExtendedMBRObject instead of MarshalByRefObject. You'll also have to remove the calls to `InitializeLifetimeService()`, as this is currently done by the superclass. You can now add the following properties for each class (all of them are optional) to the server-side configuration file:

Property	Description
<Typename>_Lifetime	Initial TTL in milliseconds, or "infinity"
<Typename>_RenewOnCallTime	Time to add to a method call in milliseconds
<Typename>_SponsorshipTimeout	Maximum time to react for sponsor objects in milliseconds

To make this example behave the same as the previous one, you can use the following server-side configuration file:

```
<configuration>
  <system.runtime.remoting>
    <application>

      <channels>
        <channel ref="http" port="5555" />
      </channels>

      <lifetime
            leaseTime="10MS"
            renewOnCallTime="10MS"
            leaseManagerPollTime = "5MS"
      />

      <service>

        <wellknown mode="Singleton"
                   type="Server.DefaultLifeTimeSingleton, Server"
                   objectUri="DefaultLifeTimeSingleton.soap" />

        <wellknown mode="Singleton"
                   type="Server.LongerLivingSingleton, Server"
                   objectUri="LongerLivingSingleton.soap" />

        <wellknown mode="Singleton"
                   type="Server.InfinitelyLivingSingleton, Server"
                   objectUri="InfinitelyLivingSingleton.soap" />

      </service>
    </application>
  </system.runtime.remoting>
  <appSettings>
     <add key="Server.LongerLivingSingleton_LifeTime" value="5000" />
     <add key="Server.LongerLivingSingleton_RenewOnCallTime" value="1000" />
     <add key="Server.InfinitelyLivingSingleton_LifeTime" value="infinity" />
  </appSettings>
</configuration>
```

When the new server is started (the client doesn't need any changes for this), you'll see the server-side output shown in Figure 7-3, which demonstrates that the changes were successful and the newly created server objects really read their lifetime settings from the configuration file.

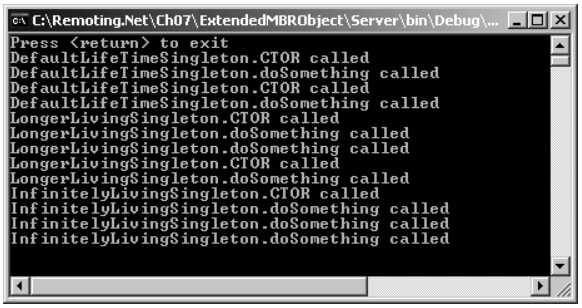

Figure 7-3. *The configured server behaves as expected.*

Working with Sponsors

Now that I've covered the primary aspects of lifetime management in the .NET Remoting framework, I next show you the probably most confusing (but also most powerful) part of it: the sponsorship concept.

Whenever a remote object is created, a sponsor can be registered with it. This sponsor is contacted by the LeaseManager as soon as the object's time to live is about to expire. It then has the option to return a TimeSpan, which will be the new TTL for the remote object. When a sponsor doesn't want to extend an object's lifetime, it can simply return TimeSpan.Zero.

The sponsor object itself is a MarshalByRefObject that has to implement the interface ISponsor. The only other requisite for a sponsor is to be reachable by the .NET Remoting framework. It can therefore be located either on the remoting server itself, on another server application, or on the client application.

■ **Caution** Be aware, though, that when using client-side sponsors, the server has to be able to contact the client directly (the client becomes a server itself in this case, as it's hosting the sponsor object). When you are dealing with clients behind firewalls, this approach will not work.

Implementing the ISponsor Interface

Sponsors have to implement the ISponsor interface, which is defined in System.Runtime.Remoting.Lifetime. It contains just one method, which will be called by the LeaseManager upon expiration of a lease's time to live.

```
public interface ISponsor
{
      TimeSpan Renewal(System.Runtime.Remoting.Lifetime.ILease lease)
}
```

The sponsor has to return a TimeSpan that specifies the new TTL for the object. If the sponsor decides not to increase the LeaseTime, it can return `TimeSpan.Zero`. A basic sponsor can look like the following:

```
public class MySponsor: MarshalByRefObject, ISponsor
{
   private bool NeedsRenewal()
   {

      // check some internal conditions

      return true;
   }

   public TimeSpan Renewal(System.Runtime.Remoting.Lifetime.ILease lease)
   {
      if (NeedsRenewal())
      {
         return TimeSpan.FromMinutes(5);
      }
      else
      {
         return TimeSpan.Zero;
      }
   }
}
```

Using Client-Side Sponsors

When using client-side sponsors, you are basically mimicking the DCOM behavior of pinging, although you have more control over the process here. After acquiring the reference to a remote object (you'll do this mostly for CAOs, as for SAOs the lifetime should normally be managed only by the server), you contact its lifetime service and register the sponsor with it.

You can get an object's LifetimeService, which will be an ILease object, using the following line of code:

```
ILease lease = (ILease) obj.GetLifetimeService();
```

■ **Note** If you used an interface-based remoting approach, you might first have to cast your proxy object to MarshalByRefObject before you can call `GetLifetimeService()`. The preceding line would in this case read `ILease lease = (ILease) ((MarshalByRefObject) obj).GetLifetimeService();`.

The ILease interface supports a `Register()` method to add another sponsor for the underlying object. When you want to hold a reference to an object for an unspecified amount of time (maybe while your client application is waiting for some user input), you can register a client-side sponsor with it and increase the TTL on demand.

Calling an Expired Object's Method

In the example shown in Listing 7-4, you see the result of calling an expired object's method. This happens because the server-side lifetime is set to one second, whereas the client uses a five-second delay between two calls to the CAO. Please note that, contrary to the previous example, I decided that I will only share two interfaces between client and server. I have used a so-called object factory approach in which the client contacts a SAO which will in turn create an instance of the requested object and hand back a reference to this explicitly created CAO. In addition, I'll use the RemotingHelper class, which I've introduced in Chapter 4.

I will use the following interfaces for this communication:

```
interface IRemoteFactory
{
  IRemoteObject CreateInstance();
}

interface IRemoteObject
{
  void DoSomething();
}
```

The output is shown in Figure 7-4.

Listing 7-4. *Catching the Exception When Calling an Expired Object*

```
using System;
using System.Runtime.Remoting;
using System.Threading;
using Server;
namespace Client
{

  class Client
  {
    static void Main(string[] args)
    {
      String filename =
             AppDomain.CurrentDomain.SetupInformation.ConfigurationFile;
      RemotingConfiguration.Configure(filename);

      IRemoteFactory fact =
             (IRemoteFactory) RemotingHelper.CreateProxy(typeof(IRemoteFactory));

      IRemoteObject cao = fact.CreateInstance();

      try
      {
        Console.WriteLine("{0} CLIENT: Calling doSomething()", DateTime.Now);
```

```
        cao.DoSomething();
      }
      catch (Exception e)
      {
        Console.WriteLine(" --> EX: Timeout in first call\n{0}",e.Message);
      }

      Console.WriteLine("{0} CLIENT: Sleeping for 5 seconds", DateTime.Now);
      Thread.Sleep(5000);

      try
      {
        Console.WriteLine("{0} CLIENT: Calling doSomething()", DateTime.Now);
        cao.DoSomething();
      }
      catch (Exception e)
      {
        Console.WriteLine(" --> EX: Timeout in second call\n{0}",e.Message );
      }

      Console.WriteLine("Finished ... press <return> to exit");
      Console.ReadLine();
      Console.ReadLine();
    }
  }
}
```

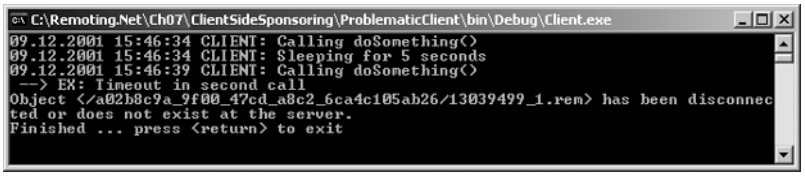

Figure 7-4. *You've been calling an expired object's method.*

■ **Note** Starting with .NET Framework 1.1, you will receive the generic message "Requested Service not found" whereas previous versions of the .NET Framework included the complete ObjectURI in the error message.

There are two ways of correcting this application's issues. First, you can simply increase the object's lifetime on the server as shown in the previous examples. In a lot of scenarios, however, you won't know which TTL will be sufficient. Just imagine an application that acquires a reference to a CAO and will only call another method of this object after waiting for user input. In this case, it might be desirable to add a client-side sponsor to your application and register it with the CAO's lease.

As the first step in enabling your application to work with client-side sponsors, you have to include a port="" attribute in the channel section of the configuration file. Without this attribute, the channel will not accept callbacks from the server.

Because you might not know which port will be available at the client, you can supply a value of 0, which allows the .NET Remoting framework to choose a free port on its own. When the sponsor is created and passed to the server, the channel information that gets passed to the remote process will contain the correct port number.

```
<configuration>
  <system.runtime.remoting>
    <application>
        <channels>
           <channel ref="http" port="0" />
        </channels>
        <!-- client entries removed -->
    </application>
  </system.runtime.remoting>
</configuration>
```

In the client's code, you then add another class that implements ISponsor. To see the exact behavior of the client-side sponsor, you might also want to add a boolean flag that indicates whether the lease time should be extended or not.

```
public class MySponsor: MarshalByRefObject, ISponsor
{

   public bool doRenewal = true;

   public TimeSpan Renewal(System.Runtime.Remoting.Lifetime.ILease lease)
   {
      Console.WriteLine("{0} SPONSOR: Renewal() called", DateTime.Now);

      if (doRenewal)
      {
         Console.WriteLine("{0} SPONSOR: Will renew (10 secs)", DateTime.Now);
         return TimeSpan.FromSeconds(10);
      }
      else
      {
         Console.WriteLine("{0} SPONSOR: Won't renew further", DateTime.Now);
         return TimeSpan.Zero;
      }
   }
}
```

In Listing 7-5 you can see a client application that registers this sponsor with the server object's lease. When the application is ready to allow the server to destroy the instance of the CAO, it will tell the sponsor to stop renewing. Normally you would call `Lease.Unregister()` instead, but in this case I want to show you that the sponsor won't be contacted further after returning `TimeSpan.Zero` to the lease manager.

Listing 7-5. *Registering the Sponsor to Avoid Premature Termination of the Object*

```
using System;
using System.Runtime.Remoting;
using System.Runtime.Remoting.Lifetime;
using System.Threading;
using Server;

class ClientApp
{
   static void Main(string[] args)
   {
      String filename =
         AppDomain.CurrentDomain.SetupInformation.ConfigurationFile;
      RemotingConfiguration.Configure(filename);

      IRemoteFactory fact =
         (IRemoteFactory) RemotingHelper.CreateProxy(typeof(IRemoteFactory));
      IRemoteObject cao = fact.CreateInstance();

      ILease le = (ILease) ((MarshalByRefObject) cao).GetLifetimeService();
      MySponsor sp = new MySponsor();
      le.Register(sp);

      try
      {
         Console.WriteLine("{0} CLIENT: Calling doSomething()", DateTime.Now);
         cao.DoSomething();
      }
      catch (Exception e)
      {
         Console.WriteLine(" --> EX: Timeout in first call\n{0}",e.Message);
      }

      Console.WriteLine("{0} CLIENT: Sleeping for 5 seconds", DateTime.Now);
      Thread.Sleep(5000);

      try
      {
         Console.WriteLine("{0} CLIENT: Calling doSomething()", DateTime.Now);
         cao.DoSomething();
      }
      catch (Exception e)
      {
         Console.WriteLine(" --> EX: Timeout in second call\n{0}",e.Message );
      }
```

```
      Console.WriteLine("{0} CLIENT: Unregistering sponsor", DateTime.Now);
      le.Unregister(sp);

      Console.WriteLine("Finished ... press <return> to exit");
      Console.ReadLine();
      Console.ReadLine();
   }
}
```

If you were to run this application as-is with the previous client-side and server-side configuration files, you would receive an exception telling you that this operation is not allowed at the current security level. This is due to a new security setting that has been introduced with version 1.1 of the .NET framework. If you want to use sponsors, you now have to specifically allow this feature in your client- and server-side configuration files. To allow it, you have to explicitly state the formatters that should be used to communicate with the other party. In addition, you have to set the attribute typeFilterLevel to "Full" for each server-side formatter.

A valid server-side configuration file could therefore look like this:

```
<configuration>
  <system.runtime.remoting>
    <application>

      <channels>
            <channel ref="http" port="5555">
               <serverProviders>
                  <formatter ref="binary" typeFilterLevel="Full" />
               </serverProviders>
               <clientProviders>
                  <formatter ref="binary" />
               </clientProviders>
            </channel>
      </channels>

      <!-- Services removed -->
    </application>
  </system.runtime.remoting>
</configuration>
```

On the client side, a similar change is necessary:

```
<configuration>
  <system.runtime.remoting>
    <application>
      <channels>
          <channel ref="http" port="0">
             <clientProviders>
                <formatter ref="binary" />
             </clientProviders>
             <serverProviders>
                <formatter ref="binary" typeFilterLevel="Full" />
```

```
            </serverProviders>
          </channel>
      </channels>
     <!-- client entries removed -->
    </application>
  </system.runtime.remoting>
</configuration>
```

When you run this application, you will see the output in Figure 7-5 at the client.

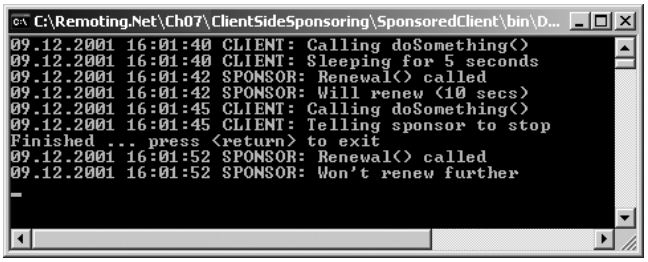
C:\Remoting.Net\Ch07\ClientSideSponsoring\SponsoredClient\bin\D...
09.12.2001 16:01:40 CLIENT: Calling doSomething()
09.12.2001 16:01:40 CLIENT: Sleeping for 5 seconds
09.12.2001 16:01:42 SPONSOR: Renewal() called
09.12.2001 16:01:42 SPONSOR: Will renew (10 secs)
09.12.2001 16:01:45 CLIENT: Calling doSomething()
09.12.2001 16:01:45 CLIENT: Telling sponsor to stop
Finished ... press <return> to exit
09.12.2001 16:01:52 SPONSOR: Renewal() called
09.12.2001 16:01:52 SPONSOR: Won't renew further

Figure 7-5. *Client-side output when using a sponsor*

As you can see in this figure, during the time the main client thread is sleeping for five seconds, the sponsor is contacted by the server. It renews the lease for another ten seconds. As soon as the client has finished its work with the remote object, it unregisters the sponsor from the lease by using the following line of code to allow the server to destroy the object:

```
le.Unregister(sponsor);
```

■ **Caution** When you decide to use client-side sponsors, you have to make sure that the client is reachable by the server. Whenever you are dealing with clients that may be located behind firewalls or proxies, you have to choose another approach!

Using Server-Side Sponsors

Server-side sponsors that are running in the same process as the target CAOs can constitute a solution to the preceding problem, but you have to keep in mind several things to make your application run stably.

First, remote sponsors are MarshalByRefObjects themselves. Therefore, they also have an assigned lifetime, and you may want to manage this yourself to provide a consistent behavior. Generally you will want your server-side sponsor to be active as long as the client application is "online." You nevertheless will have to make sure that the resources will be freed as soon as possible after the client application is ended.

One possible approach is to continuously send a command to the sponsors so that they stay alive. This can be accomplished with a simple `KeepAlive()` method that is called periodically from a background thread of the client application.

Another thing to watch out for is that the sponsor will be called from the .NET Remoting framework's LeaseManager. This call might well *increase* the time to live of your sponsor, depending on the RenewOnCallTime set in the configuration file. Without taking special care here, you might end up with sponsors that keep running forever when `Unregister()` has not been called correctly for each and every sponsored object. This could happen, for example, when the client application crashes or experiences a network disconnect.

To remove this potential problem, I suggest you add a DateTime instance variable that holds the time of the last call to the sponsor's `KeepAlive()` method. When the LeaseManager calls `Renew()`, the difference between the current time and the last time `KeepAlive()` has been called will be checked, and the sponsored object's lease will only be renewed when the interval is below a certain limit. As soon as all objects that are monitored by this sponsor are timed out, no further calls will be placed to the sponsor itself, and its own lease will therefore expire as well.

In the following example, I used very short lease times to show you the implications in greater detail. In production applications, you should probably keep this in the range of the default TTL, which is five minutes. The source code in Listing 7-6 (which has to be defined in the shared DLL) shows you the implementation of a sponsor that supports the described functionality.

Listing 7-6. *The Server-Side Sponsor That Is Pinged by the Client*

```
using System;
using System.Runtime.Remoting.Lifetime;
using System.Runtime.Remoting;
using Server; // for ExtendedMBRObject

namespace Sponsors
{
   public class InstanceSponsor: MarshalByRefObject, ISponsor
   {
      public DateTime lastKeepAlive;

      public InstanceSponsor()
      {
         Console.WriteLine("{0} SPONSOR: Created ", DateTime.Now);
         lastKeepAlive = DateTime.Now;
      }

      public void KeepAlive()
      {
         Console.WriteLine("{0} SPONSOR: KeepAlive() called", DateTime.Now);
         // tracks the time of the last keepalive call
         lastKeepAlive = DateTime.Now;
      }

      public TimeSpan Renewal(System.Runtime.Remoting.Lifetime.ILease lease)
      {
         Console.WriteLine("{0} SPONSOR: Renewal() called", DateTime.Now);
```

```
            // keepalive needs to be called at least every 5 seconds
            TimeSpan duration = DateTime.Now.Subtract(lastKeepAlive);
            if (duration.TotalSeconds < 5)
            {
               Console.WriteLine("{0} SPONSOR: Will renew (10 secs) ",
                     DateTime.Now);
               return TimeSpan.FromSeconds(10);
            }
            else
            {
               Console.WriteLine("{0} SPONSOR: Won't renew further", DateTime.Now);
               return TimeSpan.Zero;
            }
         }
      }
}
```

When implementing this concept, I have also added the following object-factory interface to the shared DLL to facilitate communication between the client and the sponsor. Its implementation will just return a new instance of the InstanceSponsor class.

```
public interface IRemoteSponsorFactory
{
   InstanceSponsor CreateSponsor();
}
```

When employing the following configuration file, the sponsor will act like this: a call to `KeepAlive()` is needed at least every five seconds (determined from the call's time of arrival at the server, so you better call it more often from your client). When this call is received, `lastKeepAlive` is set to the current time using `DateTime.Now` and (due to the RenewOnCall time set in the configuration file) its own lease time will be increased to five seconds as well.

Whenever the LeaseManager asks for a renewal, the sponsor will compare the current time to `lastKeepAlive`, and only when the difference is fewer than five seconds will it extend the sponsored object's lease.

```
<configuration>
  <system.runtime.remoting>
    <application name="SomeServer">

      <channels>
        <channel ref="http" port="5555" />
      </channels>

      <lifetime
          leaseTime="1S"
          renewOnCallTime="1S"
          leaseManagerPollTime = "100MS"
      />
```

```
      <service>
        <wellknown mode="Singleton"
                   type="Server.RemoteFactory, Server"
                   objectUri ="RemoteFactory.rem"/>
        <wellknown mode="Singleton"
                   type="Sponsors.SponsorFactory, Server"
                   objectUri ="SponsorFactory.rem"/>
      </service>

    </application>
  </system.runtime.remoting>
  <appSettings>
    <add key="Sponsors.InstanceSponsor_Lifetime" value="5000" />
    <add key="Sponsors.InstanceSponsor_RenewOnCallTime" value="5000" />
    <add key="Server.RemoteObject_Lifetime" value="4000" />
  </appSettings>
</configuration>
```

■ **Note** The preceding sample only works with the ExtendedMBRObject shown earlier in this chapter.

As this sponsor's `KeepAlive()` method needs to be called at regular intervals, you have to add another class to the client application. It will spawn a new thread that periodically calls the sponsor. This class takes an InstanceSponsor object as a constructor parameter and will call the server every three seconds until its `StopKeepAlive()` method is called.

```
class EnsureKeepAlive
{
   private bool _keepServerAlive;
   private InstanceSponsor _sponsor;

   public EnsureKeepAlive(InstanceSponsor sponsor)
   {
      _sponsor = sponsor;
      _keepServerAlive = true;
      Console.WriteLine("{0} KEEPALIVE: Starting thread()", DateTime.Now);
      Thread thrd = new Thread(new ThreadStart(this.KeepAliveThread));
      thrd.Start();
   }

   public void StopKeepAlive()
   {
      _keepServerAlive= false;
   }

   public void KeepAliveThread()
   {
```

```
      while (_keepServerAlive)
      {
         Console.WriteLine("{0} KEEPALIVE: Will KeepAlive()", DateTime.Now);
         _sponsor.KeepAlive();
         Thread.Sleep(3000);
      }
   }
}
```

To use this sponsor and its factory from a client-side project, I've added the following line to the client-side remoting configuration file:

```
  <wellknown type="Shared.IRemoteSponsorFactory, Shared"
             url="http://localhost:5555/SponsorFactory.rem" />
```

In the application itself, you have to add calls to create the server-side sponsor and to start the client-side keepalive thread:

```
IRemoteFactory fact =
     (IRemoteFactory) RemotingHelper.CreateProxy(typeof(IRemoteFactory));
IRemoteObject cao = fact.CreateInstance();

IRemoteSponsorFactory sf =
  (IRemoteSponsorFactory)
     RemotingHelper.CreateProxy(typeof(IRemoteSponsorFactory));

// create remote (server-side) sponsor
InstanceSponsor sp = sf.CreateSponsor();

// start the keepalive thread
EnsureKeepAlive keepalive = new EnsureKeepAlive(sp);

ILease le = (ILease) ((MarshalByRefObject) cao).GetLifetimeService();

// register the sponsor
le.Register(sp);

// ... rest of implementation removed
```

When you are finished using the CAO, you have to unregister the sponsor using `ILease.Unregister()` and stop the keepalive thread:

```
le.Unregister(sp);
keepalive.StopKeepAlive();
```

Even though in the preceding example I just used a single CAO, you can register this sponsor with multiple leases for different CAOs at the same time. When you run this application, you'll see the output shown in Figures 7-6 and 7-7 on the client and the server, respectively.

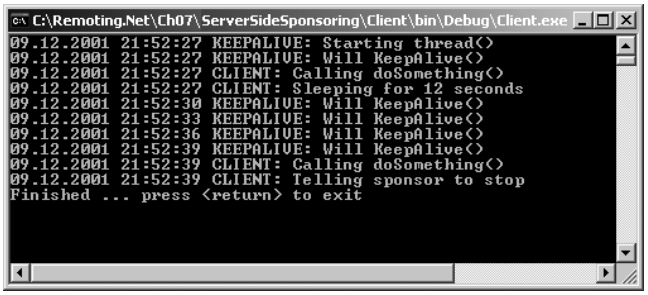
C:\Remoting.Net\Ch07\ServerSideSponsoring\Client\bin\Debug\Client.exe
09.12.2001 21:52:27 KEEPALIVE: Starting thread()
09.12.2001 21:52:27 KEEPALIVE: Will KeepAlive()
09.12.2001 21:52:27 CLIENT: Calling doSomething()
09.12.2001 21:52:27 CLIENT: Sleeping for 12 seconds
09.12.2001 21:52:30 KEEPALIVE: Will KeepAlive()
09.12.2001 21:52:33 KEEPALIVE: Will KeepAlive()
09.12.2001 21:52:36 KEEPALIVE: Will KeepAlive()
09.12.2001 21:52:39 KEEPALIVE: Will KeepAlive()
09.12.2001 21:52:39 CLIENT: Calling doSomething()
09.12.2001 21:52:39 CLIENT: Telling sponsor to stop
Finished ... press <return> to exit

Figure 7-6. *Client-side output when running with server-side sponsors*

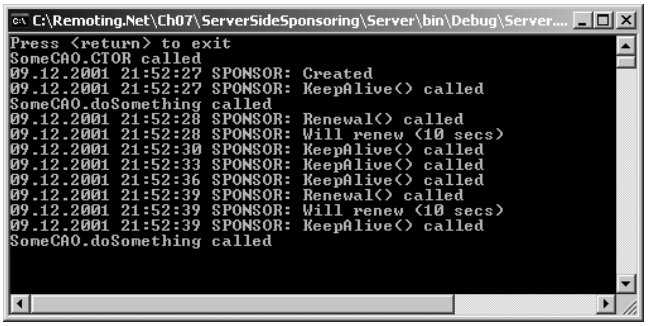
C:\Remoting.Net\Ch07\ServerSideSponsoring\Server\bin\Debug\Server....
Press <return> to exit
SomeCAO.CTOR called
09.12.2001 21:52:27 SPONSOR: Created
09.12.2001 21:52:27 SPONSOR: KeepAlive() called
SomeCAO.doSomething called
09.12.2001 21:52:28 SPONSOR: Renewal() called
09.12.2001 21:52:28 SPONSOR: Will renew (10 secs)
09.12.2001 21:52:30 SPONSOR: KeepAlive() called
09.12.2001 21:52:33 SPONSOR: KeepAlive() called
09.12.2001 21:52:36 SPONSOR: KeepAlive() called
09.12.2001 21:52:39 SPONSOR: Renewal() called
09.12.2001 21:52:39 SPONSOR: Will renew (10 secs)
09.12.2001 21:52:39 SPONSOR: KeepAlive() called
SomeCAO.doSomething called

Figure 7-7. *Server-side output when running with server-side sponsors*

In both figures, you can see that `KeepAlive()` is called several times while the client's main thread is sleeping. The server-side sponsor renews the lease two times before it's finally about to expire.

To see if the application behaves correctly when a client "crashes" while holding instances of the remote object, you can just kill the client after some calls to `KeepAlive()` and look at the server-side output, which is shown in Figure 7-8.

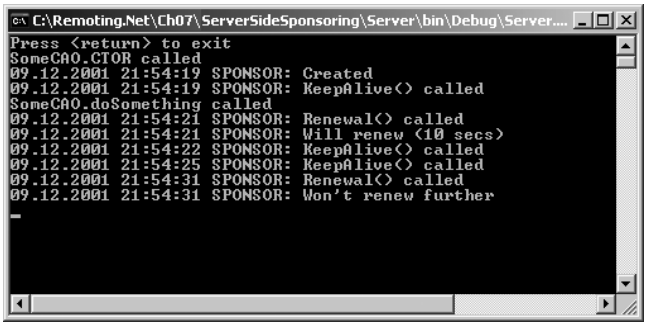
C:\Remoting.Net\Ch07\ServerSideSponsoring\Server\bin\Debug\Server....
Press <return> to exit
SomeCAO.CTOR called
09.12.2001 21:54:19 SPONSOR: Created
09.12.2001 21:54:19 SPONSOR: KeepAlive() called
SomeCAO.doSomething called
09.12.2001 21:54:21 SPONSOR: Renewal() called
09.12.2001 21:54:21 SPONSOR: Will renew (10 secs)
09.12.2001 21:54:22 SPONSOR: KeepAlive() called
09.12.2001 21:54:25 SPONSOR: KeepAlive() called
09.12.2001 21:54:31 SPONSOR: Renewal() called
09.12.2001 21:54:31 SPONSOR: Won't renew further

Figure 7-8. *Server-side output when the client is stopped during execution*

Here you can see that the sponsor processed three calls to `KeepAlive()` before the client stopped pinging. It received the call to `Renewal()` more than five seconds later than the last call to `KeepAlive()`, and therefore refused to further prolong the object's lease time. Hence you can be sure that both objects (the CAO and its sponsor) are timed out and correctly marked for garbage collection.

Using the CallContext

When creating distributed applications, one of the recurring requirements is to transfer runtime information between server and client. This is information that allows you to determine the context in which a piece of code is running.

When you call a method locally, for example, the called code will immediately know about the caller's security permissions based on the thread's security token, and based on .NET's own code access security. When you call code remotely, however, the called code cannot do a simple stack-based lookup to get access to this information. Instead, the information has to be somehow transferred to the server.

Information like this can be transferred in two different ways: as a method parameter or via some other means. The latter case is what we call *"out-of-band"* (OOB) data—information that is transferred alongside the real method call, but which helps to create the server-side context.

Some parts of these OOB data are transferred automatically by the protocol. If you host your components in IIS, for example, and activate Windows-Integrated security as discussed earlier in this book, then you don't have to manually pass username/password information. The underlying HTTP protocol and the .NET Remoting framework will automatically take care of securely transferring this OOB information for you.

In addition to security, applications can have the more generic need to pass out-of-band information between client and server. To facilitate this, the .NET Remoting framework provides the CallContext class. The call context is essentially a dictionary with a special behavior: all items that implement ILogicalThreadAffinative will travel alongside your method calls between server and client. If your client, for example, stores one of these items in the call context, then your server-side code will be able to retrieve the item from its call context.

Both the CallContext class and the interface are defined in the namespace System.Runtime.Remoting.Messaging.

An example of OOB data is to enable/disable logging based on a client-side setting. Globally enabling/disabling of server-side logging is usually not enough if you have hundreds of concurrent clients and want to isolate problems with only one of them. In this case, you can create a command-line or menu option for the client application which would then pass the necessary information to the server.

In the following application, I will again present the CustomerManager class, which I've been using throughout the book. I will extend it to include additional logging information (which is, admittedly, simulated by just using `Console.WriteLine()`). This logging information should only be written if the client has been started with the command-line switch "/enablelog". For all other concurrent clients, no information should be written to the log file.

I first need to define an object that is [Serializable] and that implements ILogicalThreadAffinative in a DLL that is shared between client and server. I use the same DLL that also contains the definition of the IRemoteCustomerManager interface and the Customer object as you can see in Listing 7-7.

Listing 7-7. *Defining the Logical Thread Affinative Objects*

```
using System;
using System.Runtime.Remoting.Messaging;

namespace General
{
   [Serializable]
   public class LogSettings: ILogicalThreadAffinative
   {
      public bool EnableLog;
   }

   public interface IRemoteCustomerManager
   {
      Customer GetCustomer(int id);
   }

   [Serializable]
   public class Customer
   {
     // implementation removed
   }
}
```

On the client side, you can now check for the command-line switch, and if necessary create a new LogSettings object and store it in the CallContext using its `SetData()` method, as shown in Listing 7-8.

Listing 7-8. *Storing the LogSettings in the CallContext*

```
static void Main(string[] args)
{
   String filename = AppDomain.CurrentDomain.SetupInformation.ConfigurationFile;
   RemotingConfiguration.Configure(filename);

   if (args.Length > 0 && args[0].ToLower() == "/enablelog")
   {
      LogSettings ls = new LogSettings();
      ls.EnableLog = true;
      CallContext.SetData("log_settings", ls);
   }

   IRemoteCustomerManager mgr =
      (IRemoteCustomerManager)
           RemotingHelper.CreateProxy(typeof(IRemoteCustomerManager));

   // the callcontext will be transferred automatically
   Customer cust = mgr.GetCustomer(42);
```

```
   Console.WriteLine("Done");
   Console.ReadLine();
}
```

■ **Note** The CallContext is scoped on the level of a thread. If your application has multiple threads, you will have independently operating call context objects.

On the server side, you can access the CallContext's data by calling `GetData()` and casting the resulting object to the corresponding type as shown in Listing 7-9.

Listing 7-9. *Accessing the CallContext on the Server Side*

```
class CustomerManager: MarshalByRefObject, IRemoteCustomerManager
{
   public Customer GetCustomer(int id)
   {
      LogSettings ls = CallContext.GetData("log_settings") as LogSettings;

      if (ls!= null && ls.EnableLog)
      {
         // simulate write to a logfile
         Console.WriteLine("LOG: Loading Customer " + id);
      }

      Customer cust = new Customer();
      return cust;
   }
}
```

If you run the application without specifying a command-line switch, you will receive the server-side output in Figure 7-9. If you however specify "/enablelog" when starting the client, the server-side output will look like what appears in Figure 7-10.

Figure 7-9. *Server-side output when client has been started without switches*

Figure 7-10. *Server-side output when client has been started with "/enablelog"*

■ **Note** You can set the command-line switches that should be used during debugging in Visual Studio .NET by right-clicking your project and navigating to Configuration Properties ➤ Debugging. The field you are looking for is called "Command Line Arguments".

As the call context is valid for more than one method call, you will have to manually remove elements that you do not need anymore (otherwise they will travel back and forth with every method call). You can do this by using `CallContext.FreeNamedDataSlot()`.

Best Practices

In your application code, you should normally avoid directly accessing the call context, as it ties your business code to the .NET Remoting framework. I usually recommend using certain strongly typed wrapper classes like the ones shown in Listing 7-10 instead. This will allow you to later (with future versions of the .NET Framework) move your implementation to a different protocol without any dependencies to the .NET Remoting framework. In this case, you will only need to change the wrapper class, which provides for a clean separation between your business logic and the transport layer.

You can define such a wrapper similar to the one in Listing 7-10.

Listing 7-10. *A Wrapper for Log Settings*

```
public class LogSettingContext
{
   public static bool EnableLog
   {
      get
      {
         LogSettings ls = CallContext.GetData("log_settings") as LogSettings;

         if (ls!= null)
         {
            return ls.EnableLog;
         }
         else
         {
            return false;
         }
```

```
      }
      set
      {
         LogSettings ls = new LogSettings();
         ls.EnableLog = value;
         CallContext.SetData("log_settings", ls);
      }
   }
}
```

You can then set the call context flag on the client side like this:

```
if (args.Length > 0 && args[0].ToLower() == "/enablelog")
{
   LogSettingContext.EnableLog=true;
}
```

and read it on the server side with code similar to the following:

```
if (LogSettingContext.EnableLog)
{
   // simulate write to a logfile
   Console.WriteLine("LOG: Loading Customer " + id);
}
```

Security and the Call Context

Please note that you should *never* use the CallContext to transfer the thread's current principal (i.e., information about the currently logged-in user) between client and server. This would be insecure, and a malicious client application could easily send whatever information it would like.

Remoting Events

With the remoting of events, you are now reaching an area of .NET Remoting where the intuitive way of solving a problem might not be the correct one.

■ **A Serious Warning** Before I show you how to work with remoting events, let me please state that these events are mostly useful for cross-application communication on a single machine. An example of such a scenario is to have a Windows service that does some background processing and that communicates with a Systray application/icon or with a conventional Windows Forms application. Remoting events *can* be used in LAN-based environments with a limited number of listeners. They will, however, *not* scale up to support hundreds of receivers or to support WAN-based environments. This is not directly a fault of the .NET Remoting system, but of the way the underlying TCP connections are used. In the next chapter, I've detailed the reasons for this and present some non-remoting-based solutions for asynchronous notifications in different environments.

Events: First Take

Let's say you have to implement a type of broadcast application in which a number of clients *(always according to the preceding warning!)* register themselves at the servers as *listeners*, and other clients can send messages that will be broadcast to all listening clients.

You need to take into account two key facts when using this kind of application. The first one is by design: when the event occurs, client and server will change roles. This means that the client in reality becomes the server (for the callback method), and the server will act as a client and try to contact the "real" client. This is shown in Figure 7-11.

■ **Caution** This implies that clients located behind firewalls are not able to receive events using any of the included channels!

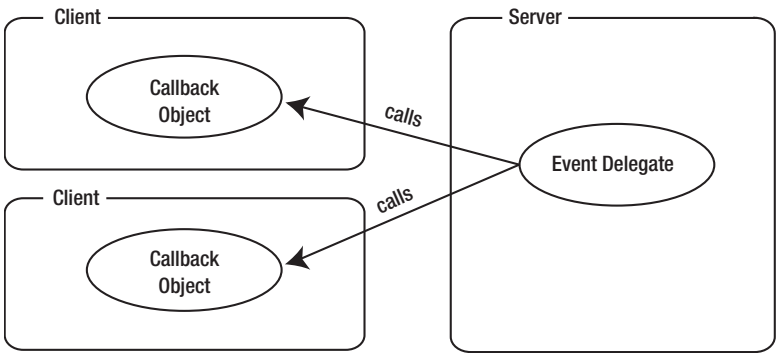

Figure 7-11. *The clients will be contacted by the server.*

The second issue can be seen when you are "intuitively" developing this application. In this case, you'd probably start with the interface definition shown in Listing 7-11, which would be compiled to General.dll and shared between the clients and the server.

Listing 7-11. *The IBroadcaster Interface (Nonworking Sample)*

```
using System;
using System.Runtime.Remoting.Messaging;

namespace General {

   public delegate void MessageArrivedHandler(String msg);

   public interface IBroadcaster {
      void BroadcastMessage(String msg);
      event MessageArrivedHandler MessageArrived;
   }
}
```

This interface allows clients to register themselves to receive a notification by using the `MessageArrived` event. When another client calls `BroadcastMessage()`, this event will be invoked and the listening clients called back. The server-side implementation of this interface is shown in Listing 7-12.

Listing 7-12. *The Server-Side Implementation of IBroadcaster*

```
using System;
using System.Runtime.Remoting;
using System.Threading;
using General;

namespace Server
{

   public class Broadcaster: MarshalByRefObject, IBroadcaster
   {

      public event General.MessageArrivedHandler MessageArrived;

      public void BroadcastMessage(string msg) {
         // call the delegate to notify all listeners
         Console.WriteLine("Will broadcast message: {0}", msg);
         MessageArrived(msg);
      }

      public override object InitializeLifetimeService() {
         // this object has to live "forever"
         return null;
      }
   }

   class ServerStartup
   {
      static void Main(string[] args)
      {
         String filename = "server.exe.config";
         RemotingConfiguration.Configure(filename);

         Console.WriteLine ("Server started, press <return> to exit.");
         Console.ReadLine();
      }
   }
}
```

The listening client's implementation would be quite straightforward in this case. The only thing you'd have to take care of is that the object that is going to be called back to handle the event has to be a MarshalByRefObject as well. This is shown in Listing 7-13.

Listing 7-13. *The First Client's Implementation, Which Won't Work*

```
using System;
using System.Runtime.Remoting;
using General;
using RemotingTools; // RemotingHelper

namespace EventListener
{
   class EventListener
   {
      static void Main(string[] args)
      {
         String filename = "eventlistener.exe.config";
         RemotingConfiguration.Configure(filename);

         IBroadcaster bcaster =
            (IBroadcaster) RemotingHelper.CreateProxy(typeof(IBroadcaster));

         Console.WriteLine("Registering event at server");

         // callbacks can only work on MarshalByRefObjects, so
         // I created a different class for this as well
         EventHandler eh = new EventHandler();
         bcaster.MessageArrived +=
            new MessageArrivedHandler(eh.HandleMessage);

         Console.WriteLine("Event registered. Waiting for messages.");
         Console.ReadLine();
      }
   }

   public class EventHandler: MarshalByRefObject {
      public void HandleMessage(String msg) {
         Console.WriteLine("Received: {0}",msg);
      }

      public override object InitializeLifetimeService() {
         // this object has to live "forever"
         return null;
      }
   }
}
```

When implementing this so-called intuitive solution, you'll be presented with the error message shown in Figure 7-12.

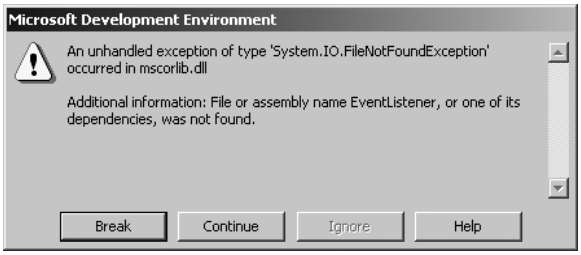
Microsoft Development Environment
An unhandled exception of type 'System.IO.FileNotFoundException' occurred in mscorlib.dll
Additional information: File or assembly name EventListener, or one of its dependencies, was not found.
Break
Continue
Ignore
Help

Figure 7-12. *An exception occurs when combining the delegate with the remote event.*

This exception occurs while the request is deserialized at the server. At this point, the delegate is restored from the serialized message, and it tries to validate the target method's signature. For this validation, the delegate attempts to load the assembly containing the destination method. In the case presented previously, this will be the client-side assembly EventListener.exe, which is not available at the server.

You're probably thinking, "Great, but how can I use events nevertheless?" I show you how in the next section.

Refactoring the Event Handling

As always, there's the relatively easy solution of shipping the delegate's destination assembly to the caller. This would nevertheless mean that the client-side application has to be referenced at the server—doesn't sound that nice, does it?

Instead, you can introduce an intermediate MarshalByRefObject (including the implementation, not only the interface) that will be located in General.dll and will therefore be accessible by both client and server.

```
public class BroadcastEventWrapper: MarshalByRefObject {
   public event MessageArrivedHandler MessageArrivedLocally;

   [OneWay]
   public void LocallyHandleMessageArrived (String msg) {
      // forward the message to the client
      MessageArrivedLocally(msg);
   }

   public override object InitializeLifetimeService() {
      // this object has to live "forever"
      return null;
   }
}
```

■ **Note** This is still not the final solution, as there are some problems with using [OneWay] events in real-world applications as well. I cover this shortly after the current example!

This wrapper is created in the client's context and provides an event that can be used to call back the "real" client. The server in turn receives a delegate to the BroadcastEventWrapper's `LocallyHandleMessageArrived()` method. This method activates the BroadcastEventWrapper's `MessageArrivedLocally` event, which will be handled by the client. You can see the sequence in Figure 7-13.

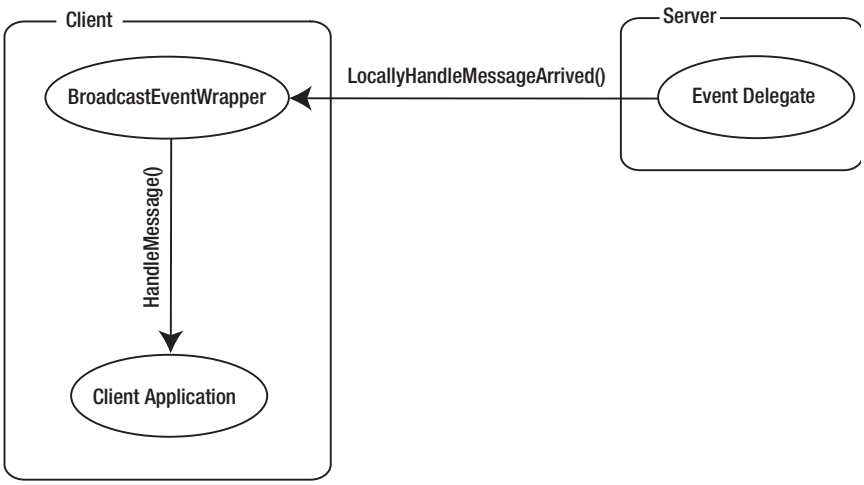

Figure 7-13. *Event handling with an intermediate wrapper*

The server's event will therefore be handled by a MarshalByRefObject that is known to it (as it's contained in General.dll) so that the delegate can resolve the method's signature. As the BroadcastEventWrapper runs in the client context, its own delegate has access to the real client-side event handler's signature.

The complete source code to General.dll is shown in Listing 7-14.

Listing 7-14. *The Shared Assembly Now Contains the BroadcastEventWrapper.*

```
using System;
using System.Runtime.Remoting.Messaging;

namespace General {

   public delegate void MessageArrivedHandler(String msg);

   public interface IBroadcaster {
      void BroadcastMessage(String msg);
      event MessageArrivedHandler MessageArrived;
   }
```

```
   public class BroadcastEventWrapper: MarshalByRefObject {
      public event MessageArrivedHandler MessageArrivedLocally;

      [OneWay]
      public void LocallyHandleMessageArrived (String msg) {
         // forward the message to the client
         MessageArrivedLocally(msg);
      }

      public override object InitializeLifetimeService() {
         // this object has to live "forever"
         return null;
      }

   }
}
```

The listening client's source code has to be changed accordingly. Instead of passing the server a delegate to its own `HandleMessage()` method, it has to create a BroadcastEventWrapper and pass the server a delegate to this object's `LocallyHandleMessageArrived()` method. The client also has to pass a delegate to its own `HandleMessage()` method (the "real" one) to the event wrapper's `MessageArrivedLocally` event.

The changed listening client's source code is shown in Listing 7-15.

Listing 7-15. *The New Listening Client's Source Code*

```
using System;
using System.Runtime.Remoting;
using General;
using RemotingTools; // RemotingHelper

namespace EventListener
{
   class EventListener
   {
      static void Main(string[] args)
      {
         String filename = "eventlistener.exe.config";
         RemotingConfiguration.Configure(filename);

         IBroadcaster bcaster =
            (IBroadcaster) RemotingHelper.CreateProxy(typeof(IBroadcaster));

         // this one will be created in the client's context and a
         // reference will be passed to the server
         BroadcastEventWrapper eventWrapper =
            new BroadcastEventWrapper();
```

```
        // register the local handler with the "remote" handler
        eventWrapper.MessageArrivedLocally +=
           new MessageArrivedHandler(HandleMessage);

        Console.WriteLine("Registering event at server");
        bcaster.MessageArrived +=
           new MessageArrivedHandler(eventWrapper.LocallyHandleMessageArrived);

        Console.WriteLine("Event registered. Waiting for messages.");
        Console.ReadLine();
     }

     public static void HandleMessage(String msg) {
        Console.WriteLine("Received: {0}",msg);
     }
  }
}
```

Before you can start this application, you also have to use a server-side configuration file that sets the typeFilterLevel to "Full" as shown here:

```
<configuration>
  <system.runtime.remoting>
    <application>
      <channels>
        <channel ref="http" port="5555">
           <serverProviders>
               <formatter ref="binary" typeFilterLevel="Full" />
           </serverProviders>
           <clientProviders>
               <formatter ref="binary" />
           </clientProviders>
        </channel>
      </channels>
      <!-- service entries removed -->
    </application>
  </system.runtime.remoting>
</configuration>
```

When this client is started, you will see the output in Figure 7-14, which shows you that the client is currently waiting for remote events. You can, of course, start an arbitrary number of clients, because the server-side event is implicitly based on a MulticastDelegate.

Figure 7-14. *The client is waiting for messages.*

To start broadcasting messages to all listening clients, you'll have to implement another client. I'm going to call this one EventInitiator in the following examples. The EventInitiator will simply connect to the server-side SAO and invoke its `BroadcastMessage()` method. You can see the complete source code for EventInitiator in Listing 7-16.

Listing 7-16. *EventInitiator Simply Calls BroadcastMessage().*

```
using System;
using System.Runtime.Remoting;
using System.Runtime.Remoting.Activation;
using General;
using RemotingTools; // RemotingHelper


namespace Client
{
   class Client
   {

      static void Main(string[] args)
      {
         String filename = "EventInitiator.exe.config";
         RemotingConfiguration.Configure(filename);

         IBroadcaster bcast =
            (IBroadcaster) RemotingHelper.CreateProxy(typeof(IBroadcaster));

         bcast.BroadcastMessage("Hello World! Events work fine now . . . ");

         Console.WriteLine("Message sent");
         Console.ReadLine();
      }
   }
}
```

When EventInitiator is started, the output shown in Figure 7-15 will be displayed at each listening client, indicating that the remote events now work as expected.

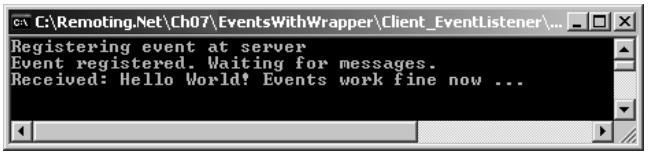
C:\Remoting.Net\Ch07\EventsWithWrapper\Client_EventListener\...
Registering event at server
Event registered. Waiting for messages.
Received: Hello World! Events work fine now ...

Figure 7-15. *Remote events now work successfully!*

Why [OneWay] Events Are a Bad Idea

You might have read in some documents and articles that remoting event handlers should be defined as [OneWay] methods. The reason is that *without* defining remote event handlers this way, an exception will occur whenever a client is unreachable or has been disconnected without first unregistering the event handler.

When just forwarding the call to your event's delegate, as shown in the previous server-side example, two things will happen: the event will not reach all listeners, and the client that initiated the event in the first place will receive an exception. This is certainly not what you want to happen.

When using [OneWay] event handlers instead, the server will try to contact each listener but won't throw an exception if it's unreachable. This seems to be a good thing at first glance. Imagine, however, that your application will run for several months without restarting. As a result, a lot of "unreachable" event handlers will end up registered, and the server will try to contact each of them every time. Not only will this take up network bandwidth, but your performance will suffer as well, as each "nonworking" call might take up some seconds, adding up to minutes of processing time for each event. This, again, is something you wouldn't want in your broadcast application.

Instead of using the default event invocation mechanism (which is fine for local applications), you will have to develop a server-side wrapper that calls all event handlers in a try/catch block and removes all nonworking handlers afterwards. This implies that you define the event handlers *without* the [OneWay] attribute! To make this work, you first have to remove this attribute from the shared assembly.

```
public class BroadcastEventWrapper: MarshalByRefObject {
   public event MessageArrivedHandler MessageArrivedLocally;

   // don't use OneWay here!
   public void LocallyHandleMessageArrived (String msg) {
      // forward the message to the client
      MessageArrivedLocally(msg);
   }

   public override object InitializeLifetimeService() {
      // this object has to live "forever"
      return null;
   }
}
```

In the server-side code, you remove the call to `MessageArrived()` and instead implement the logic shown in Listing 7-17, which iterates over the list of registered delegates and calls each one. When an exception is thrown by the framework because the destination object is unreachable, the delegate will be removed from the event.

Listing 7-17. *Invoking Each Delegate on Your Own*

```
using System;
using System.Runtime.Remoting;
using System.Threading;
using General;
```

```
namespace Server
{

   public class Broadcaster: MarshalByRefObject, IBroadcaster
   {

      public event General.MessageArrivedHandler MessageArrived;

      public void BroadcastMessage(string msg) {
         Console.WriteLine("Will broadcast message: {0}", msg);
         SafeInvokeEvent(msg);
      }

      private void SafeInvokeEvent(String msg) {
         // call the delegates manually to remove them if they aren't
         // active anymore.

         if (MessageArrived == null) {
            Console.WriteLine("No listeners");
         } else {

            Console.WriteLine("Number of Listeners: {0}",
                              MessageArrived.GetInvocationList().Length);

            MessageArrivedHandler mah=null;

            foreach (Delegate del in MessageArrived.GetInvocationList()) {
               try {
                  mah = (MessageArrivedHandler) del;
                  mah(msg);
               } catch (Exception e) {
                  Console.WriteLine("Exception occured, will remove Delegate");
                  MessageArrived -= mah;
               }
            }
         }
      }

      public override object InitializeLifetimeService() {
         // this object has to live "forever"
         return null;
      }
   }


   class ServerStartup
   {
      static void Main(string[] args)
```

```
      {
         String filename = "server.exe.config";
         RemotingConfiguration.Configure(filename);

         Console.WriteLine ("Server started, press <return> to exit.");
         Console.ReadLine();
      }
   }
}
```

When using events in this way, you ensure the best possible performance, no matter how long your server application keeps running. But you still have to keep in mind that the use of .NET Remoting events might not be the best choice to broadcast information to a large number of clients! You can read more about these scenarios in the following chapter.

Summary

In this chapter you learned about the details of .NET Remoting–based applications. You now know how lifetime is managed and how you can dynamically configure an object's time to live. If this doesn't suffice, implementing client- or server-side sponsors gives you the opportunity to manage an object's lifetime independently of any TTLs.

You also read about versioning, and you can now look at the whole application's lifecycle over various versions and know what to watch out for in regard to SAOs and CAOs, and know how the ISerializable interface can help you when using [Serializable] objects.

On the last pages of this chapter, you read about how you can use delegates and events, and what to take care of when designing an application that relies on these features. In particular, you learned that using [OneWay] event handlers the intuitive way certainly isn't the best practice.

You should now be able to solve most challenges that might confront you during design and development of a .NET Remoting application. In the next two chapters, I will share some additional tips, best practices, and troubleshooting guides that you should take into account before designing your .NET Remoting–based solution.

CHAPTER 8

The Ins and Outs of Versioning

In most distributed applications, it's of uttermost importance to look at the application's lifecycle right from the beginning. You might have to ensure that your already deployed clients will keep working, even when your server is available in newer versions and will be providing more functionality.

Generally speaking, .NET Remoting supports the base .NET versioning services, which also implies that you have to use strong names for versioning of CAOs or serializable objects, for example. Nevertheless, in details the means of lifecycle management differ quite heavily between .NET Remoting and common .NET versioning and also differ between the various types of remoteable objects.

.NET Framework Versioning Basics

Versioning itself is an integral part of the .NET Framework and the common language runtime itself. Although versioning of distributed applications is different, the basic concepts of versioning .NET Framework are still valid. Therefore, let's start with a short introduction of versioning with .NET.

Basically, each .NET component has a version attribute stored in the assembly manifest. This version consists of four parts: major and minor version as well as build number and revision number (*major.minor.build.revision*). When it comes to versioning, a general rule is that components with different major and/or minor version might be incompatible with previous versions, whereas components with the same major or minor version and different build or revision numbers leave interfaces intact to stay compatible with previous versions of the component.

The ffull assembly name consists of the simple, unencrypted assembly name, the version as described previously, the assembly culture, and a public key signature. As soon as an assembly is signed with a private key, we are talking about a so-called strong name.

A Short Introduction to Strong Naming

A *strong name* consists of the assembly's name, version, culture information, and a *fingerprint* from the publisher's public/private key pair. This scheme is used to identify an assembly "without doubt," because even though another Person could possibly create an assembly having the same name, version, and culture information, only the owner of the correct key pair can *sign* the assembly and provide the correct fingerprint. Furthermore, the signature avoids tampering of code when it gets downloaded, for example, from a Web server.

Creation of a Strongly Named Assembly

To generate a key pair to later sign your assemblies with, you have to use sn.exe with the following syntax:

```
sn.exe -k <keyfile>
```

■ **Note** All command-line tools should be accessed from the Visual Studio .NET command prompt, which is located in Start ➤ Programs ➤ Microsoft Visual Studio .NET ➤ Visual Studio.NET Tools. This version of the command prompt will set the environment variables that are needed for the use of these tools.

For example, to create a key pair that will be stored in the file mykey.key, you can run sn.exe, as shown in Figure 8-1.

VS.NET Cmd (small)
X:\>sn -k mykey.snk
Microsoft (R) .NET Framework Strong Name Utility Version 1.1.4322.573
Copyright (C) Microsoft Corporation 1998-2002. All rights reserved.
Key pair written to mykey.snk
X:\>

Figure 8-1. *Running sn.exe to generate a key pair*

■ **Caution** You *absolutely* have to keep this key secret. If someone else acquires your key, he or she can sign assemblies in your name. When using a publisher-based security scheme, this might compromise your enterprise security measures.

When you want to generate a strongly named assembly, you have to put some attributes in your source files (or update them when using VS .NET, which already includes those attributes in the file AssemblyInfo.cs, which is by default added to every project):

```
using System.Reflection;
using System.Runtime.CompilerServices;

[assembly: AssemblyCulture("")]
[assembly: AssemblyVersion("1.0.0.1")]
[assembly: AssemblyDelaySign(false)]
[assembly: AssemblyKeyFile("mykey.key")]
```

As the `AssemblyVersion` attribute defaults to "1.0.*"[1] in VS .NET, you'll have to change this to allow for definite assignment of version numbers for your components. Make sure, though, to change it whenever you distribute a new version of your DLL.

The attributeA `AssemblyKeyFile` has to point to the file generated by sn.exe. When using Visual Studio .NET, you have to place it in the directory that contains your project file (<project>.csproj for C# projects).

Upon compilation of this project, no matter whether you're using VS .NET or the command-line compilers, the keyfile will be used to sign the assembly, and you'll end up with a strongly named assembly that can be installed in the GAC.

How the CLR Locates Assemblies

When an assembly is loaded, the CLR determines all referenced assemblies through the metadata. The CLR takes the exact version of the referenced assemblies and, if available on the target machine, loads them. An application always automatically runs with the versions of referenced assemblies against which it has been compiled.[2] Therefore, the CLR needs a mechanism for locating these assemblies.

For this purpose, basically the CLR differentiates between private and shared assemblies. Private assemblies are stored in the same directory as the application using the assembly. Only other assemblies running in the same directory can use the assembly, as others won't find it.

A shared assembly can be used across multiple A shared assembly can be used across multiple assemblies stored in different directories. Shared assemblies are either stored in a shared directory[3] or in the *Global Assembly Cache* (GAC). The GAC is a central directory in the Windows system directory that can store multiple versions of the same assembly—but all assemblies in the GAC must have a strong name so that they can be identified uniquely.

If an assembly gets loaded, the CLR resolves assembly references in the following order:

1. First of all, for all sstrongly named assemblies the CLR looks into the GAC, and if the referenced assembly in the correct version can be found there, it uses this version of the assembly.

2. If no assembly has been found in the GAC, the CLR verifies whether there are any publisher policies or <codeBase> configurations available. If yes, these will be used for locating the referenced assemblies.

3. When the referenced assemblies are not found in the <codeBase> or through publisher policies, the CLR starts a so-called *probing process*. This means that the CLR tries to find the referenced assemblies within the application base directory and well-defined subdirectories (either determined by the assembly name, the culture, or through configuration). More about pprobing can be found on MSDN at `http://msdn.microsoft.com/library/default.asp?url=/library/en-us/cpguide/html/cpconHowRuntimeLocatesAssemblies.asp`.

---

1. The * in this case means that this part of the version number is assigned automatically.
2. This rule can be overridden by configuring an assembly redirection in your application configuration using <dependentAssembly> together with a <bindingRedirect> or with an assembly publisher policy through the .NET Framework configuration utility.
3. Using assemblies in a shared directory requires the <codeBase> option to be configured in the application that is using the shared assembly. This can be done in the application configuration file (app.config).

After the CLR has resolved the assembly reference based on its (full) assembly name, it can locate the referenced assemblies through the process described previously and load them. Whereas private assemblies and <codeBase> configuration basically enables *xcopy* deployment, putting an assembly into the Global Assembly Cache requires an extra installation step.

Installation in the GAC

To manipulate the contents of the GAC, you can use either Explorer to drag and drop your assemblies to %WINDOWS%\Assembly or GacUtil from the .NET Framework SDK. Here are the parameters you'll use most:

Parameter	Description
/i <assemblyname>	Installs the specified assembly in the GAC. <assemblyname> has to include the extension (.DLL or .EXE).
/l [<filter>]	Lists the contents of the GAC. If <filter> is present, only assemblies matching the filter will be listed.
/u <assembly>	Unregisters and removes a given assembly from the GAC. When <assembly> contains a *weak name* (that is, it contains only the assembly's name), *all* versions of this assembly will be uninstalled. When using a strong name, only the matching assembly will be removed from the GAC.

A Simple Versioning Example

FFor demonstrating the versioning concepts of the .NET Framework, I will walk you through a simple example. The example will consist of an assembly DLL doing some simple calculations and a client application using this component. The component will be installed in the GAC.

Afterwards, you'll see how to create a new version of the assembly component without recompiling the old client. The new version will be installed in the GAC, too. When two versions of the component exist on your machine, you can create another client using the new version. Both clients will run side by side, without any problems. Last but not least, you will configure your first client for using the newer version of the component via a binding redirect. Of course, that only works when the interface of the component will not be broken.

Start by creating a new class library project. In the project directory, create a strong name key pair for the component using `sn -k SimpleComponent.snk`. After you have created the strong name key pair, you can sign the assembly using the AssemblyKeyFile attribute, as you see in Listing 8-1.

Listing 8-1. *A Simple Component—Version 1.0*

```
using System;
using System.Reflection;

[assembly: AssemblyDelaySign(false)]
[assembly: AssemblyKeyFile(@"..\..\SimpleComponent.snk")]
[assembly: AssemblyVersion("1.0.0.0")]
```

```
namespace SimpleComponent
{
  public class SimpleClass
  {
    public SimpleClass()
    {
    }

    public string DoSomething(string a)
    {
      return string.Format("First version {0}", a);
    }
  }
}
```

When you have created and compiled the component, you install them in the global assembly cache using `gacutil /i SimpleComponent.dll`.

Next, start a new Visual Studio solution and create the first client application. This will be a console application. For referencing the previously created component, you now have to browse manually to the component DLL (SimpleComponent.dll) with the Add References dialog box. After you've added the reference, you have to set the `CopyLocal` property for the reference to false because you've installed the component in the GAC and will use the shared assembly (see Figure 8-2).

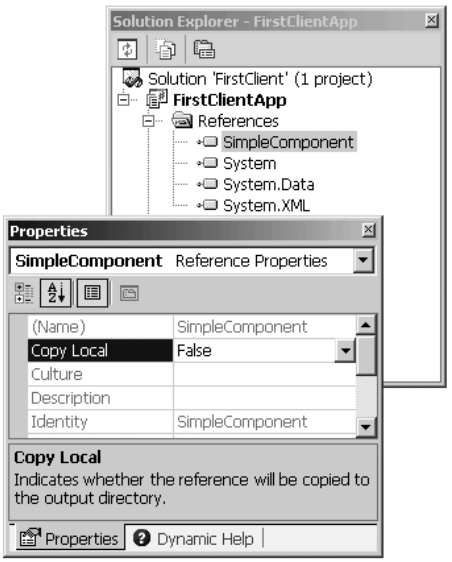

Figure 8-2. *Configuring the properties for the reference correctly*

The code for the first client is fairly simple and can be seen in Listing 8-2.

Listing 8-2. *The First Client of Your Component*

```
using System;
using SimpleComponent;

namespace FirstClientApp
{
  class ClientOne
  {
    [STAThread]
    static void Main(string[] args)
    {
      Console.WriteLine("Client 1");

      SimpleClass cls = new SimpleClass();
      Console.WriteLine(cls.DoSomething("Called from client 1"));
      Console.ReadLine();
    }
  }
}
```

Now that you have the first version of your component as well as the first client in place, you can start creating a new version of the component. For this purpose, you change the code of the component as shown in Listing 8-3.

Listing 8-3. *The Changed (Version 2) of the Component*

```
using System;
using System.Reflection;

[assembly: AssemblyDelaySign(false)]
[assembly: AssemblyKeyFile(@"..\..\SimpleComponent.snk")]
[assembly: AssemblyVersion("2.0.0.0")]

namespace SimpleComponent
{
  public class SimpleClass
  {
    public SimpleClass()
    {
    }

    public string DoSomething(string a)
    {
      return string.Format("Second version {0}", a);
    }
  }
}
```

The new version of your simple component must again be installed in the GAC. When you take a look at the installed assemblies in the GAC now, you should see two versions of your SimpleComponent class in the Global Assembly Cache, as appears in Figure 8-3.

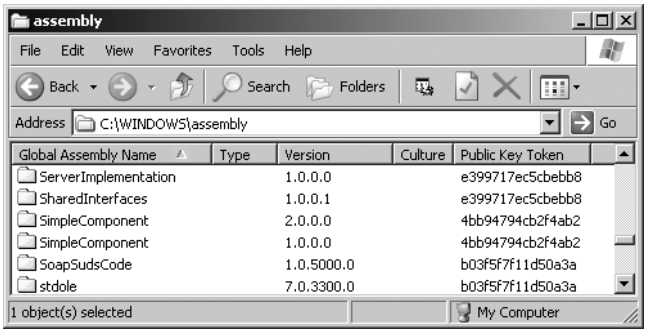

Figure 8-3. *The Global Assembly Cache with two versions of the component*

Now you can create a new client that uses the new version. For this purpose, create a new solution with Visual Studio .NET (again a console application) that looks very similar to the first version of the client as you can see in Listing 8-4.

■ **Note** After you have recompiled the second version of the SimpleComponent, don't recompile the first client, because in that case the first client will be compiled against the new version of SimpleComponent. You can avoid this by copying the different versions of SimpleComponent into separate directories after you have built them and reference them from these directories instead of directly from the bin directory of the SimpleComponent.

Listing 8-4. *The Second Client—As Simple As the First Client*

```
using System;
using SimpleComponent;

namespace SecondClientApp
{
  class SecondClientApp
  {
    [STAThread]
    static void Main(string[] args)
    {
      Console.WriteLine("Second Client started!");

      SimpleClass cls = new SimpleClass();
      string result = cls.DoSomething("Called from 2nd client...");
```

```
      Console.WriteLine("Result: " + result);
    }
  }
}
```

Figures 8-4 and 8-5 show the output of the two applications. You can see that they are using the versions of SimpleComponent they have been built with.

Figure 8-4. *The first client works with the first version.*

Figure 8-5. *The second client works with the second version.*

The last thing I want to show you is doing a binding redirect for the first client so that it uses the second version, too. For this purpose, you have to add an application configuration file for the first client (FirstClientApp.exe.config in the same directory as the application itself) with the following content:

```
<configuration>
  <runtime>
    <assemblyBinding xmlns="urn:schemas-microsoft-com:asm.v1">
      <dependentAssembly>
        <assemblyIdentity name="SimpleComponent"
                          publicKeyToken="4bb94794cb2f4ab2"
                          culture="neutral" />
        <bindingRedirect oldVersion="1.0.0.0"
                         newVersion="2.0.0.0" />
      </dependentAssembly>
    </assemblyBinding>
  </runtime>
</configuration>
```

When you have this configuration file in place, the client uses the second version of the assembly, although it has been compiled against the first version, as you can see in the assembly manifest. Figure 8-6 shows an ILDASM extract with the contained metadata (still referencing version 1.0.0.0 of SimpleComponent), whereas Figure 8-7 shows that the application is already using the new version.

Figure 8-6. *Metadata shows that FirstClient still references version 1.0.0.0.*

Figure 8-7. *The first client calls the second version because of the bindingRedirect.*

■ **Caution** Generally, the GAC should be used as infrequently as possible because of the additional management and deployment actions that are necessary. Only very general components used by most of your different applications should be installed in GAC. In other cases, it is better to deploy assemblies in the assembly directory. (Why deploy them in GAC if they are used by just one application?)

Versioning in .NET Remoting—Fundamentals

Although versioning is built into the Common Language Runtime, versioning for distributed applications is still (and will ever be) a hard challenge. Before I share some architectural and design thoughts, let's take a look at how versioning can be implemented technically with .NET Remoting.

Versioning of Server-Activated Objects

As SAOs are instantiated on demand by the server itself, there is no direct way of managing their lifecycle. The client cannot specify to which version of a given SAO a call should be placed. The

only means for supporting different versions of a SAO is to provide different URLs for them. In this case, you would have to tell your users about the new URL in other ways, as no direct support of versioning is provided in the framework.

Depending on your general architecture, you may want to place SAOs in a different assembly *or* have them in two strongly named assemblies that differ only in the version number. In the remoting configuration file, you can specify which version of a SAO is published using which URL.

Lifecycle of a Versioned SAO

Lifecycle management for a SAO becomes an issue as soon as you change some of its behavior and want currently available clients that use the older version to continue working.

In the following example, I show you how to create a SAO that's placed in a strongly named assembly. You then install the assembly in the GAC and host the SAO in IIS. The implementation of the first version 1.0.0.1, shown in Listing 8-5, returns a string that later shows you which version of the SAO has been called.

Listing 8-5. *Version 1.0.0.1 of the Server*

```
using System;
using System.Runtime.Remoting.Lifetime;
using System.Runtime.Remoting;
using System.Reflection;
using System.Runtime.CompilerServices;

[assembly: AssemblyCulture("")] // default
[assembly: AssemblyVersion("1.0.0.1")]
[assembly: AssemblyDelaySign(false)]
[assembly: AssemblyKeyFile("mykey.key")]

namespace VersionedSAO
{
   public class SomeSAO: MarshalByRefObject
   {
      public String getSAOVersion()
      {
         return "Called Version 1.0.0.1 SAO";
      }
   }
}
```

After compilation, you have to put the assembly in the GAC using gacutil.exe /i, as shown in Figure 8-8.

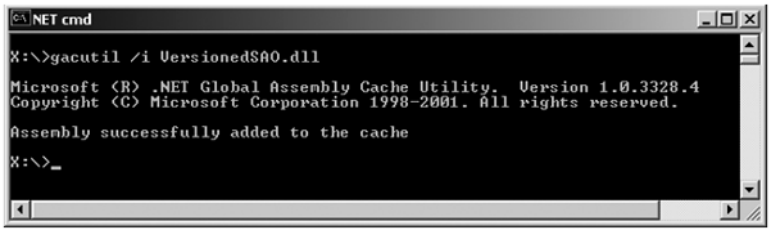
NET cmd
X:\>gacutil /i VersionedSAO.dll
Microsoft (R) .NET Global Assembly Cache Utility. Version 1.0.3328.4
Copyright (C) Microsoft Corporation 1998-2001. All rights reserved.
Assembly successfully added to the cache
X:\>

Figure 8-8. *Registering the first version in the GAC*

This DLL does not have to be placed in the /bin subdirectory of the IIS virtual directory but is instead loaded directly from the GAC. You therefore have to put the complete strong name in web.config.

You can use gacutil.exe /l <assemblyname> to get the strong name for the given assembly, as shown in Figure 8-9.

NET cmd
X:\>gacutil /l VersionedSAO
Microsoft (R) .NET Global Assembly Cache Utility. Version 1.0.3328.4
Copyright (C) Microsoft Corporation 1998-2001. All rights reserved.
The Global Assembly Cache contains the following assemblies:
VersionedSAO, Version=1.0.0.1, Culture=neutral, PublicKeyToken=84d24a897
bf5808f, Custon=null
The cache of ngen files contains the following entries:
Number of items = 1
X:\>

Figure 8-9. *Displaying the strong name for an assembly*

When editing web.config, you have to put the assembly's strong name in the type attribute of the <wellknown> entry as follows:

```
<configuration>
  <system.runtime.remoting>
    <application>
      <service>
        <wellknown mode="Singleton"
           type="VersionedSAO.SomeSAO, VersionedSAO,
                 Version=1.0.0.1,Culture=neutral,PublicKeyToken=84d24a897bf5808f"
           objectUri="MySAO.soap" />

      </service>
    </application>
  </system.runtime.remoting>
</configuration>
```

■ **Note** The type entry may only be line wrapped between the class name and the assembly's strong name—no breaks are allowed *within* the name!

Building the First Client

For the implementation of the client, you can extract the metadata using SoapSuds.

```
soapsuds-ia:VersionedSAO -nowp -oa:generated_meta_V1_0_0_1.dll
```

In the following example, I show you the implementation of a basic client that contacts the SAO and requests version information using the `getSAOVersion()` method. After setting a reference to generated_meta_V1_0_0_1.dll, you can compile the source code shown in Listing 8-6.

Listing 8-6. *Version 1.0.0.1 of the Client Application*

```
using System;
using System.Runtime.Remoting;
using System.Runtime.Remoting.Lifetime;
using System.Threading;
using VersionedSAO; // from generated_meta_xxx.dll

namespace Client
{
   class Client
   {
      static void Main(string[] args)
      {
         String filename = "client.exe.config";
         RemotingConfiguration.Configure(filename);

         SomeSAO obj = new SomeSAO();
         String result = obj.getSAOVersion();

         Console.WriteLine("Result: {0}",result);

         Console.WriteLine("Finished ... press <return> to exit");
         Console.ReadLine();
      }
   }
}
```

As the metadata assembly (generated_meta_V1_0_0_1.dll) does not have to be accessed using its strong name, the configuration file for the client looks quite similar to the previous examples.

```
<configuration>
  <system.runtime.remoting>
    <application>
```

```
        <client>
           <wellknown
              type="VersionedSAO.SomeSAO, generated_meta_V1_0_0_1"
              url="http://localhost/VersionedSAO/MySAO.soap" />
        </client>

    </application>
  </system.runtime.remoting>
</configuration>
```

When this client is started, you will see the output shown in Figure 8-10.

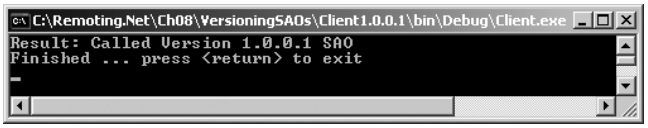

Figure 8-10. *Output of the client using the version 1.0.0.1 SAO*

Enhancing the Server

Assume you now want to improve the server with the implementation of additional application requirements that might break your existing clients. To allow them to continue working correctly, you will have to let the clients choose which version of the SAO they want to access.

In the new server's implementation, shown in Listing 8-7, you first have to change the `AssemblyVersion` attribute to reflect the new version number, and you will also want to change the server's only method to return a different result from that of the version 1.0.0.1 server.

■ **Note** When compiling the project, you will use the exact same keyfile (mykey.key) for generating the assembly's strong name!

Listing 8-7. *Version 2.0.0.1 of the Server*

```
using System;
using System.Runtime.Remoting.Lifetime;
using System.Runtime.Remoting;
using System.Reflection;
using System.Runtime.CompilerServices;

[assembly: AssemblyCulture("")] // default
[assembly: AssemblyVersion("2.0.0.1")]
[assembly: AssemblyDelaySign(false)]
[assembly: AssemblyKeyFile("mykey.key")]
```

```
namespace VersionedSAO
{
   public class SomeSAO: MarshalByRefObject
   {
      public String getSAOVersion()
      {
         return "Called Version 2.0.0.1 SAO";
      }
   }
}
```

After compiling and installing the assembly in the GAC using GacUtil, you can list the contents of the assembly cache as shown in Figure 8-11.

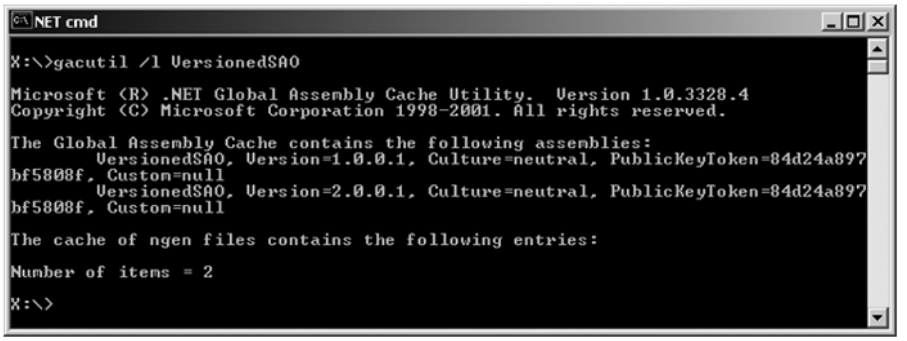
NET cmd
X:\>gacutil /l VersionedSAO
Microsoft (R) .NET Global Assembly Cache Utility. Version 1.0.3328.4
Copyright (C) Microsoft Corporation 1998-2001. All rights reserved.
The Global Assembly Cache contains the following assemblies:
VersionedSAO, Version=1.0.0.1, Culture=neutral, PublicKeyToken=84d24a897
bf5808f, Custom=null
VersionedSAO, Version=2.0.0.1, Culture=neutral, PublicKeyToken=84d24a897
bf5808f, Custom=null
The cache of ngen files contains the following entries:
Number of items = 2
X:\>

Figure 8-11. *GAC contents after installing the second assembly*

To allow a client to connect to either the old or the new assembly, you have to include a new <wellknown> entry in web.config that also points to the newly created SAO and uses a different URL.

```
<configuration>
  <system.runtime.remoting>
    <application>
      <service>

        <wellknown mode="Singleton"
           type="VersionedSAO.SomeSAO, VersionedSAO,
                 Version=1.0.0.1,Culture=neutral,PublicKeyToken=84d24a897bf5808f"
           objectUri="MySAO.soap" />

        <wellknown mode="Singleton"
           type="VersionedSAO.SomeSAO, VersionedSAO,
                 Version=2.0.0.1,Culture=neutral,PublicKeyToken=84d24a897bf5808f"
           objectUri="MySAO_V2.soap" />
```

```
      </service>
    </application>
  </system.runtime.remoting>
</configuration>
```

Developing the New Client

To allow a client application to access the second version of the SAO, you again have to generate the necessary metadata using SoapSuds.

```
soapsuds -ia:VersionedSAO -nowp -oa:generated_meta_V2_0_0_1.dll
```

After adding the reference to the newly generated metadata assembly, you also have to change the client-side configuration file to point to the new URL.

```
<configuration>
  <system.runtime.remoting>
    <application>
         <client>
                <wellknown
                          type="VersionedSAO.SomeSAO, generated_meta_V2_0_0_1"
                          url="http://localhost/VersionedSAO/MySAO_V2.soap" />
        </client>

    </application>
  </system.runtime.remoting>
</configuration>
```

You can now start both the new and the old client to get the outputs shown in Figure 8-12 for the first version and in Figure 8-13 for the second.

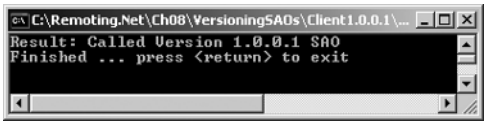

Figure 8-12. *Version 1 client running*

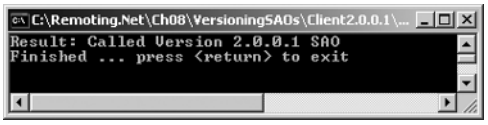

Figure 8-13. *Version 2 client running*

Both clients are running side by side at the same time, accessing the same physical server. You can also see that there was no change needed to the first client, which is the primary requisite for consistent lifecycle management.

Versioning of Client-Activated Objects

Now that you know about the lifecycle management issues with SAOs, I have to tell you that versioning of CAOs is completely different. But first, let's start with a more general look at the creation of client-activated objects.

When CAOs are instantiated by the client (using the `new` operator or `Activator.CreateInstance`), a ConstructionCallMessage is sent to the server. In this message, the client passes the name of the object it wants to be created to the server-side process. It also includes the strong name (if available) of the assembly in which the server-side object is located. This version information is stored in the [SoapType()] attribute of the SoapSuds-generated assembly. SoapSuds does this automatically whenever the assembly, passed to it with the -ia parameter, is strongly named.

Let's have a look at the C# source shown in Listing 8-8, which is generated by `soapsuds -ia -nowp -gc` from a simplistic CAO. I've inserted several line breaks to enhance its readability.

Listing 8-8. *The SoapSuds-Generated Nonwrapped Proxy's Source*

```
using System;
using System.Runtime.Remoting.Messaging;
using System.Runtime.Remoting.Metadata;
using System.Runtime.Remoting.Metadata.W3cXsd2001;
namespace Server {

[Serializable,
SoapType(SoapOptions=SoapOption.Option1|
                  SoapOption.AlwaysIncludeTypes|SoapOption.XsdString|
                  SoapOption.EmbedAll,

XmlNamespace="http://schemas.microsoft.com/clr/nsassem/Server/Server%2C%20 ➥
Version%3D2.0.0.1%2C%20Culture%3Dneutral%2C%20PublicKeyToken%3D84d24a897bf ➥
5808f",
XmlTypeNamespace="http://schemas.microsoft.com/clr/nsassem/Server/Server%2 ➥
C%20Version%3D2.0.0.1%2C%20Culture%3Dneutral%2C%20PublicKeyToken%3D84d24a8 ➥
97bf5808f")]

    public class SomeCAO : System.MarshalByRefObject
    {

   [SoapMethod(SoapAction=
   "http://schemas.microsoft.com/clr/nsassem/Server.SomeCAO/Server#doSomething")]
        public void doSomething()
        {
            return;
        }

    }
}
```

The strings in the `XmlNamespace` and `XmlTypeNamespace` attributes are URLEncoded variants of the standard version information. In plain text, they read as follows (omitting the base namespace):

```
Server, Version=2.0.0.1, Culture=neutral, PublicKeyToken= 84d24a897bf5808f
```

Doesn't look that scary anymore? In fact, this is the common .NET representation of a strong name as seen before.

What you can see now is that this proxy assembly will reference a server-side object called Server.SomeCAO, which is located in the assembly server with the strong name shown previously. Whenever a client creates a remote instance of this CAO, the server will try to instantiate the exact version of this type.

What the server does when the requested version is not available is to take the *highest* version of the specified assembly. When versions 1.0.1.0 and 2.0.0.1 are available in the GAC, and version 1.0.0.1 is requested, the server will choose 2.0.0.1 to instantiate the requested object—*even though the versions differ in the major version number.*

■ **Note** This behavior differs from the standard .NET versioning approach, in which the highest version with the same major and minor version is chosen.

To emulate the standard behavior for resolving assembly versions, or to redirect to a completely different version, you can use the assemblyBinding entry in the application's configuration file.

```
<configuration>
  <system.runtime.remoting>
    <application name="SomeServer">

      <channels>
        <channel ref="http" port="5555" />
      </channels>

      <service>
        <activated type="Server.SomeCAO, Server" />
      </service>

    </application>
  </system.runtime.remoting>
  <runtime>
   <assemblyBinding xmlns="urn:schemas-microsoft-com:asm.v1">
      <dependentAssembly>
         <assemblyIdentity name="server"
                publicKeyToken="84d24a897bf5808f"
                culture="neutral" />
                <bindingRedirect oldVersion="1.0.0.1"
                newVersion="1.0.1.1" />
      </dependentAssembly>
    </assemblyBinding>
  </runtime>
</configuration>
```

In this case, the server will take any requests for version 1.0.0.1 and use version 1.0.1.1 instead. Remember that this only works when the assembly is registered in the GAC and that you have to use `soapsuds -ia:<assembly> -nowp -oa:<meta.dll>` for each server-side version, as the [SoapType()] attribute defines this behavior.

Versioning of [Serializable] Objects

Because a [Serializable] object is marshaled by value and its data is passed as a copy, versioning behavior is once more different from SAOs or CAOs. First let's again have a look at the transfer format of the Customer object (and not the complete message) from a server similar to the one in the first example in Chapter 1.

```
<a1:Customer id="ref-4" xmlns:a1="http://schemas.microsoft.com/clr/nsassem/ ➥
VersionedSerializableObjects/VersionedSerializableObjects%2C%20Version%3D1. ➥
0.0.1%2C%20Culture%3Dneutral%2C%20PublicKeyToken%3D84d24a897bf5808f">
<FirstName id="ref-5">John</FirstName>
<LastName id="ref-6">Doe</LastName>
<DateOfBirth>1950-12-12T00:00:00.0000000+01:00</DateOfBirth>
</a1:Customer>
```

AAs you can see here, the complete namespace information, including the assembly's strong name, is sent over the wire. When the client that fetched this Customer object using a statement like `Customer cust = CustomerManager.getCustomer(42)` does not have access to this *exact* version, a SerializationException ("Parse Error, no assembly associated with Xml key") will be thrown.

To enable a "one-way relaxed" versioning schema, you can include the attribute `includeVersions = "false"` in the formatter's configuration entry as shown here:

```
<configuration>
  <system.runtime.remoting>
    <application name="SomeServer">
      <channels>
        <channel ref="http" port="5555">
          <serverProviders>
             <formatter ref="soap" includeVersions="false"/>
          </serverProviders>
        </channel>
      </channels>
    </application>
  </system.runtime.remoting>
</configuration>
```

After this change, the server will return a different serialized form of the object, which does not contain the assembly's strong name.

The newly returned Customer object's data will look like this:

```
<a1:Customer id="ref-4"  ➥
xmlns:a1="http://schemas.microsoft.com/clr/nsassem/VersionedSerializable/ ➥
Objects/VersionedSerializableObjects">
<FirstName id="ref-5">John</FirstName>
```

```
<LastName id="ref-6">Doe</LastName>
<DateOfBirth>1950-12-12T00:00:00.0000000+01:00</DateOfBirth>
</a1:Customer>
```

This last step, however, has not yet solved all issues with versioned [Serializable] objects. Let's get back to the original need for versioning in the first place: functionality is added to an application, and you want the currently available clients to keep working. This leads to the question of what will happen when you add another property to either the client or the server side's shared assembly (in the example, I'll use `public String Title` for the property). The Customer class now looks like this:

```
 [Serializable]
public class Customer
{
   public String FirstName;
   public String LastName;
   public DateTime DateOfBirth;
   public String Title; // new!
}
```

When the new Customer object (let's call it version 2.0.0.1 or just version 2 for short) is available at the client, and the old object (1.0.0.1, or just version 1, without the `Title` property) at the server, the client is able to complete the call to `Customer cust = CustomerManager.getCustomer(42)`. The client simply ignores the fact that the server did not send a value for the Customer object's `Title` property.

It won't work the other way though. When the server has version 2 of the Customer object and the client only has version 1, a SerializationException ("Member name 'VersionedSerializableObjects.Customer Title' not found") will be thrown when the client tries to interpret the server's response. This is exactly what you wanted to avoid. To work around these limitations, you have to have a look at the ISerializable interface, which allows you to specify custom serialization methods.

```
public interface ISerializable
{
    void GetObjectData(SerializationInfo info, StreamingContext context);
}
```

When implementing ISerializable, you simply have to call the SerializationInfo object's `AddValue()` method for each field you want to include in the serialized form of the current object.

To serialize the Customer object's properties from version 1 of the preceding example (without the `Title` property), you can do the following:

```
public void GetObjectData(SerializationInfo info, StreamingContext context) {
   info.AddValue("FirstName",FirstName);
   info.AddValue("LastName",LastName);
   info.AddValue("DateOfBirth",DateOfBirth);
}
```

In addition to this implementation of `GetObjectData()`, you have to provide a special constructor for your object that takes a SerializationInfo and a StreamingContext object as parameters.

```
public Customer (SerializationInfo info, StreamingContext context) {
   FirstName = info.GetString("FirstName");
   LastName = info.GetString("LastName");
   DateOfBirth = info.GetDateTime("DateOfBirth");
}
```

This constructor is called whenever a stream that contains a Customer object is about to be deserialized.

■ **Note** It's also possible to include nested objects when using ISerializable. In this case, you have to call `info.GetValue(String name, Type type)` and cast the result to the correct type. All of those additional objects have to be [Serializable], implement ISerializable, or be MarshalByRefObjects as well.

You can see version 1 of the Customer object, which is now implemented using the ISerializable interface, in Listing 8-9.

Listing 8-9. *The First Version of the Serializable Object*

```
using System;
using System.Runtime.Serialization;

namespace VersionedSerializableObjects
{
   [Serializable]
   public class Customer: ISerializable
   {
      public String FirstName;
      public String LastName;
      public DateTime DateOfBirth;

      public Customer (SerializationInfo info, StreamingContext context)
      {
         FirstName = info.GetString("FirstName");
         LastName = info.GetString("LastName");
         DateOfBirth = info.GetDateTime("DateOfBirth");
      }

      public void GetObjectData(SerializationInfo info,
                    StreamingContext context)
      {
         info.AddValue("FirstName",FirstName);
         info.AddValue("LastName",LastName);
         info.AddValue("DateOfBirth",DateOfBirth);
      }
   }
}
```

When the fields of this object have to be extended to include a `Title` property, as in the preceding example, you have to adopt `GetObjectData()` and the special constructor.

In the constructor, you have to enclose the access to the newly added property in a try/catch block. This enables you to react to a missing value, which might occur when the remote application is still working with version 1 of the object.

In Listing 8-10, the value of the Customer object's `Title` property is set to "n/a" when the SerializationInfo object does not contain this property in serialized form.

Listing 8-10. *Manual Serialization Allows More Sophisticated Versioning*

```
using System;
using System.Runtime.Serialization;

namespace VersionedSerializableObjects {
   [Serializable]
   public class Customer: ISerializable {
      public String FirstName;
      public String LastName;
      public DateTime DateOfBirth;
      public String Title;

      public Customer (SerializationInfo info, StreamingContext context) {
         FirstName = info.GetString("FirstName");
         LastName = info.GetString("LastName");
         DateOfBirth = info.GetDateTime("DateOfBirth");
         try {
            Title = info.GetString("Title");
         } catch (Exception e) {
            Title = "n/a";
         }
      }

      public void GetObjectData(SerializationInfo info,
          StreamingContext context)
      {
         info.AddValue("FirstName",FirstName);
         info.AddValue("LastName",LastName);
         info.AddValue("DateOfBirth",DateOfBirth);
         info.AddValue("Title",Title);
      }

   }
}
```

Using this serialization technique will ensure that you can match compatible server and client versions without breaking any existing applications.

Advanced Versioning Concepts

When performing versioning, you have to take several parts of your application into consideration. Of course, you have to think of your server-side implementation and the interfaces of the remoteable objects, as well as the messages sent across the wire between the client and the server.

All these different aspects require some different ideas in terms of application design, which have to be taken into consideration from the very first version of the application. When it comes to server-side implementation, you have to think about the possibility that a change in the server's implementation breaks client applications. But I think that this happens only under specific circumstances, for example, if the client is dependent on the state of the server's object. The easiest way to avoid this is to not use stateful objects on the server—for both ease of implementation as well as scalability (think of load balancing).

Much more often interfaces on the server's objects have to be broken, or the format of messages sent across the wire will change over time. These are really hard problems that have to be solved when versioning distributed applications, because in large enterprise applications you have to make sure that even old client applications that won't be updated for any reason will continue working with your application.

This basically means you still have to support the old interfaces on your server application, and you still have to understand old serializable messages (I will go into detail about this issue later in this chapter).

Versioning with Interfaces

In my opinion, the best way for versioning is definitely through interfaces. If you think of interfaces serving as the contract between clients and the server, this represents the most explicit way of versioning. A new version of the server is exposed through new interfaces, either completely new or inherited from the first version. Of course, if necessary the old interfaces still stay valid to support old clients.

IIn this way, you can avoid having several URLs for the different versions of SAO objects and avoid additional management and deployment tasks. Let's go through an example to examine how versioning through interfaces looks and how it differs from the way of versioning SAO objects through different URLs as you've seen it in the previous chapter.

Start with the first version of your shared assembly, which defines the first version of the interfaces for using your server. Listing 8-11 shows the code of the assembly.

Listing 8-11. *The First Version of the Shared Assembly*

```
using System;
using System.Reflection;
using System.Runtime.CompilerServices;

[assembly: AssemblyTitle("Shared Assembly")]
[assembly: AssemblyVersion("1.0.0.0")]
[assembly: AssemblyDelaySign(false)]
[assembly: AssemblyKeyFile(@"..\..\..\Server.snk")]

namespace General
{
  public interface IRemoteFactory
```

```
  {
    int GetAge();
    Person GetPerson();
    void UploadPerson(Person p);
  }

  [Serializable]
  public class Person
  {
    public int Age;
    public string Firstname, Lastname;

    public Person(string first, string last, int age)
    {
      this.Age = age;
      this.Firstname = first;
      this.Lastname = last;
    }
  }
}
```

The server component will be a server-activated object that supports three operations: getting the current age, getting a Person, and uploading a Person. The shared library uses the strong name key pair of the server for signing the assembly. Remember that strong names are a good idea for versioning, although versioning with interfaces does not require strong names (but you will see how to extend the sample later in terms of changing the serialized type and then strong naming is necessary).

The first version of the server's implementation can be seen in Listing 8-12. It just implements the preceding interface.

Listing 8-12. *The First Version of the Server*

```
using System;
using System.Runtime.Remoting;
using System.Reflection;
using System.Runtime.CompilerServices;

using General;

[assembly: AssemblyTitle("Server Assembly")]
[assembly: AssemblyVersion("1.0.0.0")]
[assembly: AssemblyDelaySign(false)]
[assembly: AssemblyKeyFile(@"..\..\..\Server.snk")]

namespace Server
{
  public class ServerImpl : MarshalByRefObject, IRemoteFactory
  {
    private int _ageCount = 10;
```

```
    public int GetAge()
    {
      Console.WriteLine(">> GetAge {0}", _ageCount);
      return _ageCount;
    }

    public Person GetPerson()
    {
      Console.WriteLine(">> GetPerson()");
      Console.WriteLine(">> Returning person {0}...", _ageCount);

      Person p = new Person("Test", "App", _ageCount++);
      return p;
    }

    public void UploadPerson(Person p)
    {
      Console.WriteLine(">> UploadPerson()");
      Console.WriteLine(">> Person {0} {1} {2}", p.Firstname, p.Lastname, p.Age);

      _ageCount += p.Age;
    }
  }

  class ServerApp
  {
    [STAThread]
    static void Main(string[] args)
    {
      Console.WriteLine("Starting server...");
      RemotingConfiguration.Configure("Server.exe.config");

      Console.WriteLine("Server configured, waiting for requests!");
      System.Console.ReadLine();
    }
  }
}
```

The server itself is configured as a server-activated object of type Singleton, which can be reached through the binary formatter and a TCP channel, as you can see in the following code snippet. Remember that you are trying to upload custom strongly named types. Therefore, you have to adjust the typeFilterLevel to allow deserialization of such types.

```
<?xml version="1.0" encoding="utf-8" ?>
<configuration>
  <system.runtime.remoting>
    <application>
      <channels>
        <channel ref="tcp" port="1234">
```

```
        <serverProviders>
          <formatter ref="binary" typeFilterLevel="Full" />
        </serverProviders>
      </channel>
    </channels>
    <service>
      <wellknown type="Server.ServerImpl, Server"
                 objectUri="MyServer.rem"
                 mode="Singleton" />
    </service>
  </application>
 </system.runtime.remoting>
</configuration>
```

■ **Caution** Setting the typeFilterLevel tto full as you can see in the preceding code snippet means that you need strong authentication and encryption to secure your .NET Remoting component. You can find more about security in Chapter 5.

The first version of the client just references the shared library created previously and calls several methods on the server. This client will not be touched anymore later on. This means it is going to play the role of the client that should continue working after changing the interfaces on the server's implementation (see Listing 8-13).

Listing 8-13. *The Client Using the First Version of the Server*

```
using System;
using System.Runtime.Remoting;
using System.Runtime.Remoting.Activation;
using System.Reflection;
using System.Runtime.CompilerServices;

using General;
using General.Client;

[assembly: AssemblyVersion("1.0.0.0")]
[assembly: AssemblyDelaySign(false)]
[assembly: AssemblyKeyFile(@"..\..\..\Client.snk")]

namespace ConsoleClient
{
  class ClientApp
  {
    [STAThread]
    static void Main(string[] args)
    {
      Console.WriteLine("Configuring client...");
      RemotingConfiguration.Configure("ConsoleClient.exe.config");
```

```
      Console.WriteLine("Creating proxy...");
      IRemoteFactory factory =
                (IRemoteFactory)RemotingHelper.CreateProxy(
                                        typeof(IRemoteFactory));

      Console.WriteLine("Calling GetAge()...");
      int age = factory.GetAge();
      Console.WriteLine(">> Call successful: " + age.ToString());

      Console.WriteLine("Calling GetPerson()...");
      Person p = factory.GetPerson();
      Console.WriteLine(">> Person retrieved: {0} {1}, {2}",
                           p.Firstname, p.Lastname, p.Age.ToString());

      Console.WriteLine("Calling UploadPerson()...");
      factory.UploadPerson(new Person("Upload", "Test", 20));
      Console.WriteLine(">> Upload called successfully!");

      Console.ReadLine();
    }
  }
}
```

Although not necessary, the client is signed with a strong name key pair. The configuration for the client looks as follows:

```
<configuration>
  <system.runtime.remoting>
    <application name="FirstClient">
      <channels>
        <channel ref="tcp">
          <clientProviders>
            <formatter ref="binary" />
          </clientProviders>
        </channel>
      </channels>
      <client>
        <wellknown type="General.IRemoteFactory, General"
                   url="tcp://localhost:1234/MyServer.rem" />
      </client>
    </application>
  </system.runtime.remoting>
</configuration>
```

Now that you have created both the client and the server, you can test them. The output of the two applications is shown in Figures 8-14 and 8-15.

Figure 8-14. *The first version of the client in action*

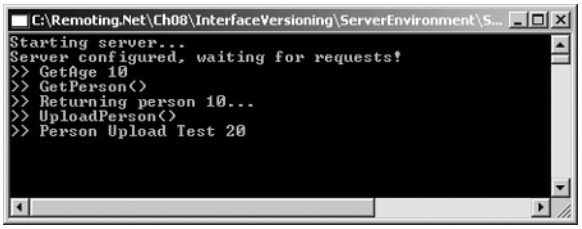

Figure 8-15. *The first version of the server in action*

Now assume that the requirements for new clients are changing—a new method called `SSetAge()` is necessary. But your old client should still be able to work with the server. Therefore, you have to leave the old interfaces intact. The solution is fairly easy: you will create a new shared assembly for the new clients where you define an interface that inherits from the base interface, implement this interface on the second version of your server, and use this interface from your second client to call the server. Listing 8-14 shows the content of the new shared library for the new clients.

Listing 8-14. *New Shared Library for the Next Generation of Your Clients*

```
using System;
using System.Reflection;
using System.Runtime.CompilerServices;

[assembly: AssemblyTitle("Shared Assembly")]
[assembly: AssemblyVersion("2.0.0.0")]

[assembly: AssemblyDelaySign(false)]
[assembly: AssemblyKeyFile(@"..\..\..\Server.snk")]

namespace GeneralV2
{
  public interface IRemoteFactory2 : General.IRemoteFactory
  {
    void SetAge(int age);
  }
}
```

Of course, the shared library has to reference the old version of your shared library—General.dll—to be able to extend the old interface. This also means new clients have to reference both the old and the new thshared assembly to be able to call the old methods, too. If interfaces and types are completely independent of the old interfaces, of course, this isn't necessary.

The second version of your server, shown in Listing 8-15, now implements the new interface, which actually inherits from the old interface. Therefore, you are not breaking compatibility and still supporting old clients.

Listing 8-15. *The New Version of Your Server*

```
using System;
using System.Runtime.Remoting;
using System.Reflection;
using System.Runtime.CompilerServices;

using General;
using GeneralV2;

[assembly: AssemblyTitle("Server Assembly")]
[assembly: AssemblyVersion("2.0.0.0")]
[assembly: AssemblyKeyFile(@"..\..\..\Server.snk")]

namespace Server
{
  public class ServerImpl : MarshalByRefObject, IRemoteFactory2
  {
    private int _ageCount = 10;

    public void SetAge(int age)
    {
      _ageCount += age;
      Console.WriteLine(">> SetAge {0}", _ageCount);
    }

    public int GetAge()
    {
      Console.WriteLine(">> GetAge {0}", _ageCount);
      return _ageCount;
    }

    public Person GetPerson()
    {
      Console.WriteLine(">> GetPerson()");
      Console.WriteLine(">> Returning person {0}...", _ageCount);

      Person p = new Person("Test", "App", _ageCount++);
      return p;
    }
```

```
    public void UploadPerson(Person p)
    {
      Console.WriteLine(">> UploadPerson()");
      Console.WriteLine(">> Person {0} {1} {2}", p.Firstname, p.Lastname, p.Age);

      _ageCount += p.Age;
    }
  }

  class ServerApp
  {
    [STAThread]
    static void Main(string[] args)
    {
      Console.WriteLine("Starting server...");
      RemotingConfiguration.Configure("Server.exe.config");

      Console.WriteLine("Server configured, waiting for requests!");
      System.Console.ReadLine();
    }
  }
}
```

In this case it's quite easy—as the new interface inherits from the old one and just adds new functionality, you have ensured that old clients are still supported. If an interface would really break the signature of existing methods, you cannot create an interface that inherits from the old version. In this case, you have to create a completely new interface that defines the contract for the new clients.

But this still means you are avoiding changes in configuration of your server components, making version management and updating server components in any case (not only new interfaces but also bug fixes and patching) much easier than deploying multiple versions of your server component under different URLs.

The second client just references the two shared assemblies (General.dll and GeneralV2.dll) and uses the new interface for calling the remoting server, as you can see in Listing 8-16.

Listing 8-16. *The Implementation of the Second Client*

```
using System;
using System.Runtime.Remoting;

using General;
using GeneralV2;
using General.Client;

namespace ClientConsole2
{
  class Class1
  {
```

```
    [STAThread]
    static void Main(string[] args)
    {
      Console.WriteLine("Configuring client...");
      RemotingConfiguration.Configure("ClientConsole2.exe.config");

      Console.WriteLine("Creating proxy...");
      IRemoteFactory2 factory =
               (IRemoteFactory2)RemotingHelper.CreateProxy(
                                        typeof(IRemoteFactory2));

      Console.WriteLine("Calling GetAge()...");
      int age = factory.GetAge();
      Console.WriteLine(">> Call successful: " + age.ToString());

      Console.WriteLine("Calling SetAge()...");
      factory.SetAge(age * 2);
      Console.WriteLine(">> Call successful!");

      Console.WriteLine("Calling GetPerson()...");
      Person p = factory.GetPerson();
      Console.WriteLine(">> Person retrieved: {0} {1}, {2}",
                        p.Firstname, p.Lastname, p.Age.ToString());

      Console.ReadLine();
    }
  }
}
```

The second client performs quite the same tasks as the first one. But it imports the namespace of both shared assemblies, General and GeneralV2, and in addition it calls the `SetAge()` function, which has been added with the second version of the interface. The configuration of the second client looks very similar to the first one despite the fact that you are using the new interface in your well-known client configuration.

```
<configuration>
  <system.runtime.remoting>
    <application name="SecondClient">
      <channels>
        <channel ref="tcp">
          <clientProviders>
            <formatter ref="binary" />
          </clientProviders>
        </channel>
      </channels>
      <client>
        <wellknown type="GeneralV2.IRemoteFactory2, GeneralV2"
                   url="tcp://localhost:1234/MyServer.rem" />
```

```
      </client>
    </application>
  </system.runtime.remoting>
</configuration>
```

Now you can run all three applications. The output is shown in Figures 8-16, 8-17, and 8-18.

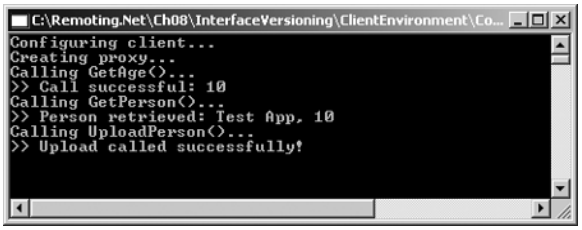

Figure 8-16. *The old client in action—still works*

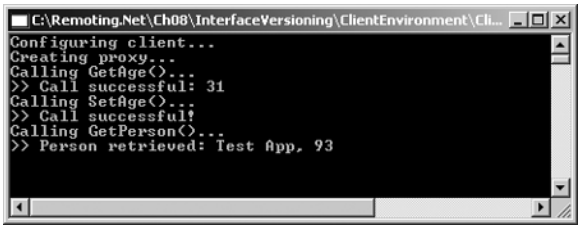

Figure 8-17. *The new client in action—leverages new functionality*

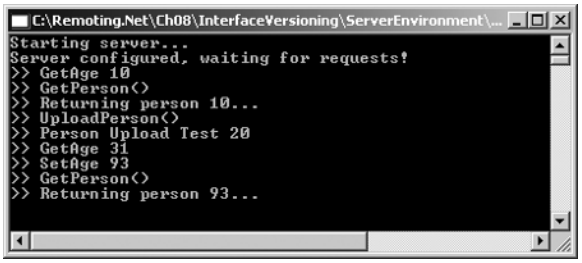

Figure 8-18. *The new version of the server after both clients have submitted requests*

Basically the interface-based approach cannot be adapted to client-activated objects directly because CAO works with an activation message sent by the client. This request includes the necessary version information for instantiating the right object on the server as you have seen earlier in this chapter.

But if you need an instance of a MarshalByRef object per client on the server, you can solve this problem differently by implementing a factory object as a Singleton running on the server, which instantiates the client-activated objects and returns them to the client. In this case, the interface-based approach works again, and you can start versioning the contract of your client-activated objects through interfaces.

Versioning Concepts for Serialized Types

Earlier in this chapter, you have been introduced to a way for versioning [Serializable] objects. The things you have seen there will be the foundation for the concepts you are using now. But what do you have to keep in mind when versioning objects sent serialized across the wire? Well, if you have only a couple of clients talking directly to the server, versioning is easier and can be done as described previously in the section "Versioning of [Serializable] Objects."

But think about a situation in which servers are exchanging data and different servers require different versions of the serializable object exchanged. What if server A works with version 2 of an object and sends this object across the wire to server B, which requires version 1? You have already seen how to solve this problem easily, but if server B sends the object across the wire to another server, server C, that understands version 2 and requires the data from the original message sent by server A, you have a problem. With the original strategy on deserialization of the extended version, the additional data of version 2 is lost on server B in the middle. Server B sends the message without the new data to server C, and therefore information for server C is missing. The scenario is described in Figure 8-19.

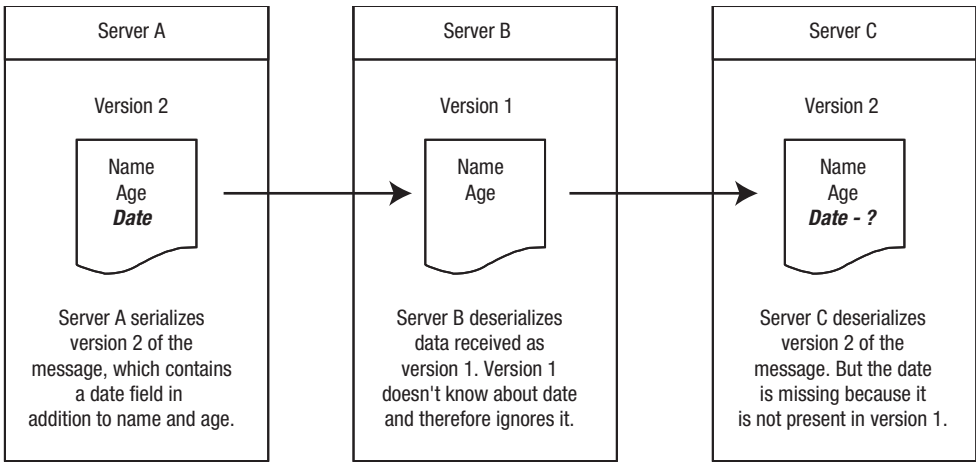
Server A
Version 2
Name
Age
Date
Server A serializes version 2 of the message, which contains a date field in addition to name and age.
Server B
Version 1
Name
Age
Server B deserializes data received as version 1. Version 1 doesn't know about date and therefore ignores it.
Server C
Version 2
Name
Age
Date - ?
Server C deserializes version 2 of the message. But the date is missing because it is not present in version 1.

Figure 8-19. *An extended versioning scenario*

Such a situation requires you to have versioning in mind from the very first moment of your application development process. Therefore, if the probability of getting in such a situation is high, you should think about the concepts being discussed in this section.

■ **Very Important** The concepts I am introducing in this chapter should only be used if they are really necessary—this means a situation like the one described previously can potentially become reality. In any other cases, avoid the additional effort of implementing versioning in this way. Furthermore, **in such situations, Web Services might be more appropriate than .NET Remoting solutions. My primary intention with this chapter is to show you that versioning of serializable types is not easy and needs to be kept in mind from the very first moment of the application design and development process!**

■ **Note** The next generation of messaging runtime, codename I*ndigo*, has such concepts built into the infrastructure through data contracts. This means with Indigo you really can concentrate on your data contract design and not—as is happening here—digging into some details of runtime serialization.

In the next example, you'll see how to implement a similar situation to the one described in Figure 8-19. Your starting position will be the first version of the client and the server from the previous example. This time you'll add an intermediary remoting server, and the client will communicate with this server instead of the original server. As you know that you are going to version your message sent across the wire—the Person object introduced in the shared library in Listing 8-11 of the first version of your interface versioning sample—you are going to change this class a little bit. Modify the Person object to support custom serialization as follows:

```
[Serializable]
public class Person : ISerializable
{
  public int Age;
  public string Firstname;
  public string Lastname;

  public Person(string first, string last, int age)
  {
    this.Age = age;
    this.Firstname = first;
    this.Lastname = last;
  }

  public Person(SerializationInfo info, StreamingContext context)
  {
    Age = info.GetInt32("Age");
    Firstname = info.GetString("Firstname");
    Lastname = info.GetString("Lastname");
  }

  public void GetObjectData(SerializationInfo info, StreamingContext context)
  {
    info.AddValue("Age", Age);
```

```
      info.AddValue("Firstname", Firstname);
      info.AddValue("Lastname", Lastname);
    }
  }
```

This will be the first version of your Person object. You'll see later on that this kind of serialization will not be sufficient for scenarios described at the beginning of this chapter. But for the moment, you will leave it to see what the problem is. Next, create your intermediary server. For simplicity, the intermediary implements the same interface as your final back-end server and just routes messages from the client to the server. The complete code for the intermediary server can be seen in Listing 8-17.

Listing 8-17. *The Intermediary Server*

```
using System;
using System.Runtime.Remoting;
using System.Reflection;
using System.Runtime.CompilerServices;

using General;
using General.Client;

[assembly: AssemblyTitle("Intermediary Server Assembly")]
[assembly: AssemblyVersion("1.0.0.0")]
[assembly: AssemblyKeyFile("")]

namespace IntermedServer
{
  public class IntermedImpl : MarshalByRefObject, IRemoteFactory
  {
    private IRemoteFactory _server;

    public IntermedImpl()
    {
      _server = (IRemoteFactory)RemotingHelper.CreateProxy(
                                    typeof(IRemoteFactory));
    }

    public int GetAge()
    {
      Console.WriteLine(">> Routing GetAge()...");
      int ret = _server.GetAge();
      Console.WriteLine(">>>> GetAge() returned {0}", ret);
      return ret;
    }

    public Person GetPerson()
    {
      Console.WriteLine(">> Routing GetPerson()...");
```

```
      Person p = _server.GetPerson();
      Console.WriteLine(">>>> GetPerson() returned {0} {1} {2}",
        p.Firstname, p.Lastname, p.Age);
      return p;
    }

    public void UploadPerson(Person p)
    {
      Console.WriteLine(">> Routing UploadPerson()...");
      _server.UploadPerson(p);
      Console.WriteLine(">>>> UploadPerson() routed successfully");
    }
  }

  class IntermedApp
  {
    [STAThread]
    static void Main(string[] args)
    {
      Console.WriteLine("Starting intermediary...");
      RemotingConfiguration.Configure("IntermedServer.exe.config");

      Console.WriteLine("Intermediary configured, waiting for requests!");
      System.Console.ReadLine();
    }
  }
}
```

The intermediary is configured to listen on port 1235 for the object URI MyIntermed.rem, as you can see in the following configuration code. Furthermore, the intermediary is configured as a client for your original server.

```
<configuration>
  <system.runtime.remoting>
    <application>
      <channels>
        <channel ref="tcp" port="1235">
          <serverProviders>
            <formatter ref="binary" typeFilterLevel="Full" />
          </serverProviders>
        </channel>
      </channels>
      <service>
        <wellknown type="IntermedServer.IntermedImpl, IntermedServer"
                   objectUri="MyIntermed.rem"
                   mode="Singleton" />
      </service>
      <client>
```

```
        <wellknown type="General.IRemoteFactory, General"
                   url="tcp://localhost:1234/MyServer.rem" />
      </client>
    </application>
  </system.runtime.remoting>
</configuration>
```

On the client you don't need to change any code. All you have to do is change the well-known client configuration to use the intermediary instead of the back-end server as you see in the following snippet:

```
<wellknown type="General.IRemoteFactory, General"
           url="tcp://localhost:1235/MyIntermed.rem" />
```

If you now run all three programs, starting the server first, then the intermediary, and last but not least the client, the output of the client and the server will not change, while the output of the intermediary does, as can be seen in Figure 8-20.

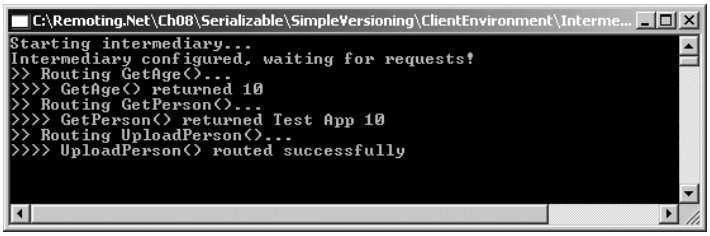

Figure 8-20. *The output of the intermediary server*

Now create a new version of your Person object with two additional properties, birth date and additional comments. The new version will be used only by the client and the back-end server. The intermediary still uses the old version of the Person object.

■ **Tip** For testing the applications with the different versions of the shared assembly, I have created a separate directory for each version of the shared assembly. Each application references the assembly from the directory where the corresponding version of the shared assembly is located. This means for this sample I have created a subdirectory, GeneralV1, where I have copied the shared assembly from the version you have seen earlier. Both the client and the intermediary server references the assembly from this directory instead of project references or directly referencing from the General's bin directory. For the second version, I have created another subdirectory, GeneralV2, where I have put the second version of the assembly. The intermediary server still references the version from the directory GeneralV1, whereas the client will know to reference the version from the directory GeneralV2.

The new version of the shared assembly, version 2.0.0.2, adds those two properties to the Person class as you can see in the following code snippet. Don't forget to change the version number in the assembly.

```
[Serializable]
public class Person : ISerializable
{
  public int Age;
  public string Firstname;
  public string Lastname;
  public DateTime Birthdate;
  public string Comments;

  public Person(string first, string last, int age)
  {
    this.Age = age;
    this.Firstname = first;
    this.Lastname = last;
  }

  public Person(SerializationInfo info, StreamingContext context)
  {
    Age = info.GetInt32("Age");
    Firstname = info.GetString("Firstname");
    Lastname = info.GetString("Lastname");

    try
    {
      Birthdate = info.GetDateTime("Birthdate");
      Comments = info.GetString("Comments");
    }
    catch { }
  }

  public void GetObjectData(SerializationInfo info, StreamingContext context)
  {
    info.AddValue("Age", Age);
    info.AddValue("Firstname", Firstname);
    info.AddValue("Lastname", Lastname);
    info.AddValue("Birthdate", Birthdate);
    info.AddValue("Comments", Comments);
  }
}
```

The implementation of the Person object exactly follows the strategy you've already seen in the section "Versioning of [Serializable] Objects" earlier in this chapter. On serialization, it just adds the new information, whereas on deserialization it tries to retrieve the new information, but if it fails it just leaves it empty. Of course, the client and the back-end server must be modified to initialize these properties. For example, the server's `GetPerson()` and `UploadPerson()` methods could be modified as follows:

```
public Person GetPerson()
{
  Console.WriteLine(">> GetPerson()");
  Console.WriteLine(">> Returning person {0}...", _ageCount);

  Person p = new Person("Test", "App", _ageCount++);
  p.Birthdate = DateTime.Now;
  p.Comments = "Try to find this info on the client!";

  return p;
}

public void UploadPerson(Person p)
{
  Console.WriteLine(">> UploadPerson()");
  Console.WriteLine(">> Person {0} {1} {2}", p.Firstname, p.Lastname, p.Age);
  Console.WriteLine(">>>> Additional properties {0} {1}",
                                        p.Birthdate, p.Comments);

  _ageCount += p.Age;
}
```

Of course, the original client's code must read or initialize the properties, too. Therefore, the code of the client could be modified as demonstrated in the following code snippet:

```
Console.WriteLine("Calling GetPerson()...");
Person p = factory.GetPerson();
Console.WriteLine(">> Person retrieved: {0} {1}, {2}",
                        p.Firstname, p.Lastname, p.Age.ToString());
Console.WriteLine(">>>> Additional properties: {0} {1}",
                        p.Birthdate, p.Comments);

Console.WriteLine("Calling UploadPerson()...");
Person up = new Person("Upload", "Test", 20);
up.Birthdate = DateTime.Now.AddDays(2);
up.Comments = "Person. Two days older.";
factory.UploadPerson(up);
Console.WriteLine(">> Upload called successfully!");
```

If you now let the client talk directly to the back-end server by configuring the well-known entry in the configuration of the client with the port and URL for the back-end server (`<wellknown type="General.IRemoteFactory, General" url="tcp://localhost:1234/MyServer.rem" />`) and test your solutions, you will see that the new properties are accepted as shown in Figures 8-21 and 8-22.

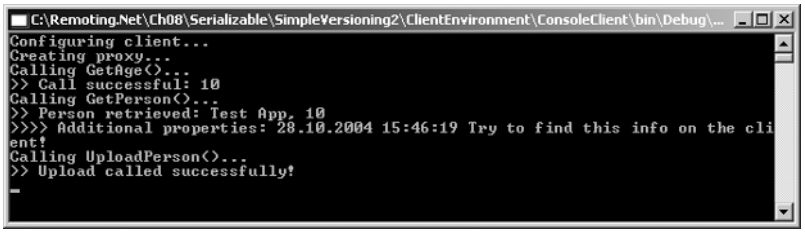

Figure 8-21. *The client with the new version of Person directly talking to the server*

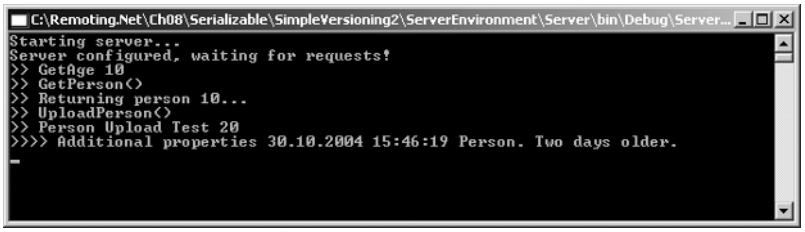

Figure 8-22. *The server with the new version of Person called directly from the client*

Now you'll change the client to use the intermediary server by again changing the well-known client configuration in its configuration file to call to port 1235 and to the server MyIntermed.rem as in the original situation. If you take a look at the running applications now, you'll recognize that any additional data is lost when sent through the intermediary. The reason for this should be obvious: the intermediary still works with version 1 of your Person class and therefore ignores the additional information on the deserialization process. And, of course, when it serializes the Person object for sending to the back-end server, it just serializes the data known from version 1. Therefore, the output of the three applications looks like what you see in Figures 8-23, 8-24, and 8-25.

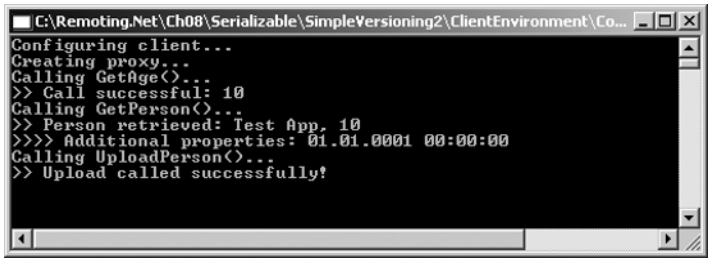

Figure 8-23. *The client with the new version of Person calling the intermediary*

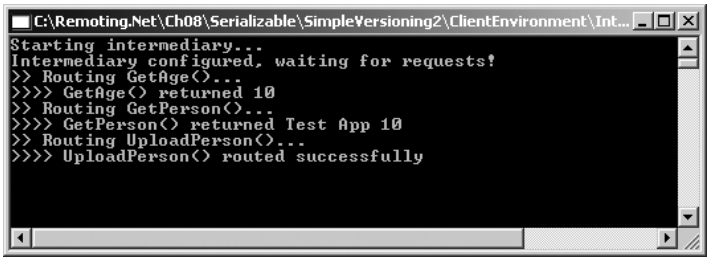

Figure 8-24. *The intermediary retrieving and sending messages of the old version*

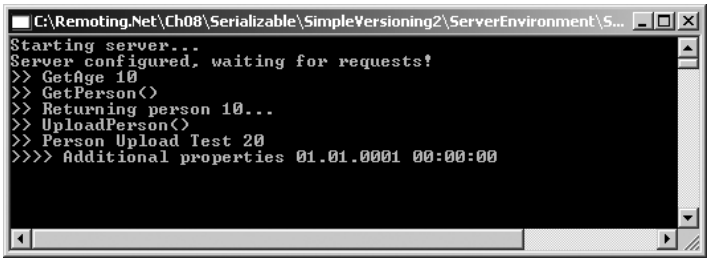

Figure 8-25. *The server with the new version of Person called from the intermediary*

■ **Note** To make the flow between the three participants with different versions of the message possible, I had to configure the option `includeVersions="false"` for all three participants. If this option is configured to true, you would get an exception either in the intermediary or in your client and server. The reason is that the runtime tries to bind serialized data always to its original version of the type. This means if your client serializes version 2.0 of Person, then the serialization runtime tries to find version 2.0 on the intermediary. But the intermediary doesn't know anything about version 2.0, and therefore the runtime throws an exception. And even if the intermediary knows about the type (for example, because it is installed in GAC), you would get an exception because the intermediary is not built for using the newer version.

Now you've seen the problem. The comment as well as the birth date are lost (take a look at the output—no comment and an empty default date). But how can you solve it? Well, the preceding sample is a typical example of having not known the versioning requirements for the application and therefore implementing a simplistic versioning strategy. The only way to fix this problem is fixing both version 1 and version 2 of the shared assembly. So what you have to do is step back to your original version—version 1 of the client, the intermediary, and the back-end server—and restart your implementation again.

The requirements of the original scenario say that messages must be able to be sent to intermediaries not understanding the newer version without losing any data. Therefore, you have to implement your very first version of Person with that in mind. If the first version retrieves information from the serialization stream that it doesn't understand, it has to still keep this information. The changed implementation of your *first version* for Person looks like the code in Listing 8-18.

Listing 8-18. *The New Shared Assembly for the First Version*

```
using System;
using System.Collections;
using System.Reflection;
using System.Runtime.Serialization;
using System.Runtime.CompilerServices;

[assembly: AssemblyTitle("Shared Assembly")]
[assembly: AssemblyVersion("1.0.0.20")]
[assembly: AssemblyKeyFile(@"..\..\..\Server.snk")]

namespace General
{
  public interface IRemoteFactory
  {
    int GetAge();
    Person GetPerson();
    void UploadPerson(Person p);
  }

  [Serializable]
  public class Person : ISerializable
  {
    public int Age;
    public string Firstname;
    public string Lastname;
    private ArrayList Reserved=null;

    public Person(string first, string last, int age)
    {
      this.Age = age;
      this.Firstname = first;
      this.Lastname = last;
    }

    public Person(SerializationInfo info, StreamingContext context)
    {
      ArrayList values = (ArrayList)info.GetValue(
                          "personData", typeof(ArrayList));

      this.Age = (int)values[0];
      this.Firstname = (string)values[1];
      this.Lastname = (string)values[2];

      Console.WriteLine("[Person]: Deserialized person: {0} {1} {2}",
                                          Firstname, Lastname, Age);
```

```
      if(values.Count > 3)
      {
        Console.WriteLine("[Person]: Found additional values...");

        Reserved = new ArrayList();
        for(int i=3; i < values.Count; i++)
          Reserved.Add(values[i]);

        Console.WriteLine("[Person]: Additional values saved!");
      }
    }

    public void GetObjectData(SerializationInfo info, StreamingContext context)
    {
      ArrayList data = new ArrayList();

      Console.WriteLine("[Person]: serializing data...");
      data.Add(Age);
      data.Add(Firstname);
      data.Add(Lastname);

      if(Reserved != null)
      {
        Console.WriteLine("[Person]: storing unknown data...");

        foreach(object obj in Reserved)
          data.Add(obj);
      }

      info.AddValue("personData", data, typeof(ArrayList));
    }
  }
}
```

As you can see in this listing, you need to change the custom serialization and deserialization quite heavily. Now the Person uses an ArrayList for storing its own data as well as additional data. All data objects that are not understood bye the Person class are stored in a private object array Reserved.

More exactly, in the special constructor of the Person class, all information understood by this version is read from the ArrayList stored in the SerializationInfo and directly assigned to the Person's members. If there is any further information present, it will be stored in a private object array called Reserved. If the runtime serializes the Person object by calling the `ISerializable.GetObjectData()` method, it serializes its own data at first and then—if present—any data from the reserved field (which is not understood by this version), too. Therefore, you don't lose any data when a newer version introduces additional fields.

■ **Warning** The implementation of such serialization ffunctions can get very hard. In this sample it's very simple. The implementation assumes that future versions of the Person class never change the serialization order of the old members! If you do so, this code will fail. Also, if you remove members in future versions, your old version will be broken. You can either develop more complex serialization logic or keep it as simple as possible (which I would suggest) for supporting your scenarios.

■ **Warning** This solution furthermore does not consider whether fields are required or not in the serialization process. Imagine the following situation: you add a third version of your Person instance that removes a field from the Person that was mandatory for older versions (version 1 or version 2) of Person. In this case, you have broken compatibility, too.

Now that you have made the changes, you can run the first versions of your sample. The output doesn't differ from the first version despite the fact that I've added some additional Console.WriteLine statements to see when and how serialization and deserialization of the Person takes place. Take a look at Figures 8-26, 8-27, and 8-28.

Figure 8-26. *The new version 1 client calling the intermediary*

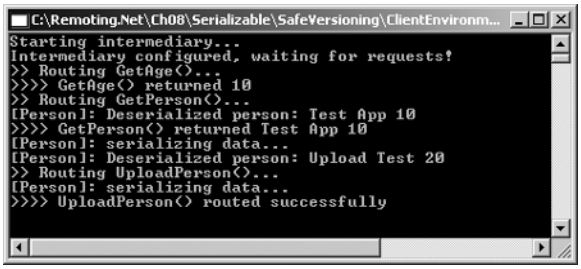

Figure 8-27. *The new version 1 intermediary using the new version 1 of Person*

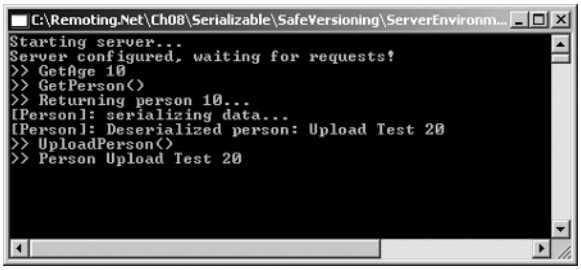

Figure 8-28. *The server called by the intermediary using the new version 1 of Person*

When you take a look at Figure 8-28, the new version 1 of your server using this new implementation of Person, you can see the two lines written by the Person class to the console. The first one, "[Person]: serializing data", is the serialization of Person by the runtime when returning Person as a result of the `GetPerson()` method from the server. For clarification, here is the code of this method on the server:

```
public Person GetPerson()
{
  Console.WriteLine(">> GetPerson()");
  Console.WriteLine(">> Returning person {0}...", _ageCount);

  Person p = new Person("Test", "App", _ageCount++);
  return p;
}
```

The second output from the Person class, "[Person]: Deserialized Person: Upload Test 20", is when the Person object gets deserialized when it is received from the client. For clarity, here is the code of the `UploadPerson()` method on the server:

```
public void UploadPerson(Person p)
{
  Console.WriteLine(">> UploadPerson()");
  Console.WriteLine(">> Person {0} {1} {2}", p.Firstname, p.Lastname, p.Age);

  _ageCount += p.Age;
}
```

You can see that neither the `GetPerson()` nor the `UploadPerson()` method outputs something like the text mentioned previously. That happens when the Person object gets serialized or deserialized by the runtime and when the runtime calls your special constructor on deserialization (before calling `UploadPerson()`) and serialization (after `GetPerson()` returns the new Person instance).

Now you can create version 2 of your shared library, client, and server and extend your Person object again adding the two additional fields as you did before. The intermediary will still use the old version of Person and will not be changed (therefore, you will keep its reference to the first version of the general assembly). Take a look at the following code, which outlines the changes in the Person object for version 2 of your shared library:

```
[Serializable]
public class Person : ISerializable
{
  public int Age;
  public string Firstname;
  public string Lastname;
  public DateTime Birthdate;    // new !!
  public string Comments;       // new !!
  private ArrayList Reserved=null;

  public Person(string first, string last, int age)
  {
    this.Age = age;
    this.Firstname = first;
    this.Lastname = last;
  }

  public Person(SerializationInfo info, StreamingContext context)
  {
    ArrayList values = (ArrayList)info.GetValue("personData", typeof(ArrayList));

    this.Age = (int)values[0];
    this.Firstname = (string)values[1];
    this.Lastname = (string)values[2];

    try
    {
      if(values.Count >= 5)
      {
        this.Birthdate = (DateTime)values[3];
        this.Comments = (string)values[4];
      }
    }
    catch { }

    Console.WriteLine("[Person]: Deserialized person: {0} {1} {2}",
                                      Firstname, Lastname, Age);

    if(values.Count > 5)
    {
      Console.WriteLine("[Person]: Found additional values...");

      Reserved = new ArrayList();
      for(int i=5; i < values.Count; i++)
        Reserved.Add(values[i]);
```

```
      Console.WriteLine("[Person]: Additional values saved!");
    }
  }

  public void GetObjectData(SerializationInfo info, StreamingContext context)
  {
    ArrayList data = new ArrayList();

    Console.WriteLine("[Person]: serializing data...");
    data.Add(Age);
    data.Add(Firstname);
    data.Add(Lastname);
    data.Add(Birthdate);
    data.Add(Comments);

    if(Reserved != null)
    {
      Console.WriteLine("[Person]: storing unknown data...");

      foreach(object obj in Reserved)
        data.Add(obj);
    }

    info.AddValue("personData", data, typeof(ArrayList));
  }
}
```

Let's start with examining the deserialization of the Person object in the special constructor. Because the new object has new information, it tries to get this information from the serialization context—of course with the two new fields. Any further information is stored in the private field Reserved as with the first version of Person. The difference here is that it assumes that the first five (instead of the first three) elements belong to itself and the rest constitute some unknown additional information.

The serialization of the object works similarly to the first version, too. It just adds the two new fields to the serialized ArrayList and then adds any additional unknown information to the ArrayList.

■ **Note** I have experienced some problems with Visual Studio .NET updating the references to the new version of my shared assembly throughout all the samples. It sometimes has not updated the reference on my client application with the new version of the assembly. Therefore, I'd suggest always changing version numbers in the assembly and verifying whether the reference uses the right version of the shared assembly.

Now that you have the new version in place, you can modify the client and the server as you already did before. In `GetPerson()`on the server, you add some code to initialize the new properties, whereas in `UploadMethod()` on the server you output the new properties as you can see in the following code snippet:

```
public Person GetPerson()
{
  Console.WriteLine(">> GetPerson()");
  Console.WriteLine(">> Returning person {0}...", _ageCount);

  Person p = new Person("Test", "App", _ageCount++);
  p.Birthdate = DateTime.Now;
  p.Comments = "Additional properties";
  return p;
}

public void UploadPerson(Person p)
{
  Console.WriteLine(">> UploadPerson()");
  Console.WriteLine(">> Person {0} {1} {2}", p.Firstname, p.Lastname, p.Age);
  Console.WriteLine(">>>> New properties {0} {1}", p.Birthdate, p.Comments);

  _ageCount += p.Age;
}
```

Of course, your client must initialize the properties so that you receive some useful values on the server. The following code snippet shows the additional modifications in the client application:

```
Console.WriteLine("Calling GetPerson()...");
Person p = factory.GetPerson();
Console.WriteLine(">> Person retrieved: {0} {1}, {2}",
                      p.Firstname, p.Lastname, p.Age.ToString());
Console.WriteLine(">>>> New properties: {0} {1}", p.Birthdate, p.Comments);

Console.WriteLine("Calling UploadPerson()...");
Person up = new Person("Upload", "Test", 20);
up.Birthdate = DateTime.Now.AddDays(2);
up.Comments = "Two days older person!";
factory.UploadPerson(up);
Console.WriteLine(">> Upload called successfully!");
```

The intermediary will not be changed in any way. It still uses the old version of Person, and you don't need to change anything in its code. When you now run the application and test the serialization behavior, you will get the output shown in Figures 8-29, 8-30, and 8-31.

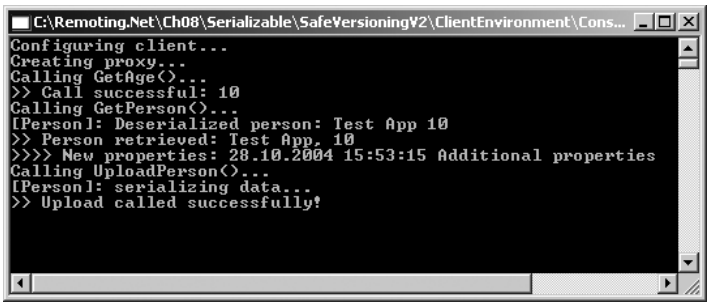

Figure 8-29. *The new version 2 client calling the intermediary using the new Person object*

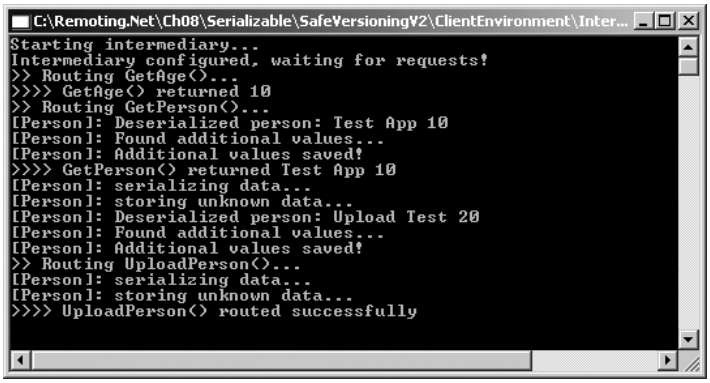

Figure 8-30. *The new version 1 intermediary using your new version 1 of Personn*

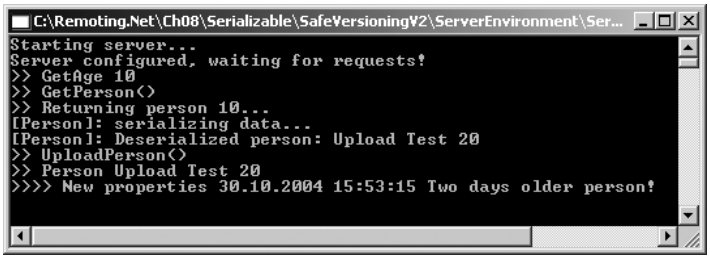

Figure 8-31. *The server called by the intermediary but itself using version 2 of Person*

As you can see in the preceding figures, the client and the back-end server understand the new information. The interesting part is the output of the intermediary, which still uses the first version of the Person class. For the first version of Person, the additional information of birth date and comments are unknown. Therefore, it serializes these new fields in its additional Reserved data field. Person always outputs information when that happens. This information can be seen in Figure 8-30 of the intermediary (look at messages like “Found additional values” or “Storing unknown data”).

With those changes in your custom serialization logic in the serializable Person class, you support the scenario of sending newer versions to remoting objects supporting older versions only, without losing any additional information.

■ **Security Warning** If data added to newer versions of the serializable type should not be readable by older versions, you have to encrypt them during serialization as well as decrypt them when deserializing the data. With this, only the newer version will be able to read potentially confidential data.

Summary

In the last sections, you have explored the details for versioning .NET components in general as well as versioning .NET Remoting components in distributed application scenarios. Independent of how you are implementing it technically, versioning must be kept in mind from the very first moment you start designing and developing the first version of your application.

Although .NET Remoting offers you several possibilities for versioning server-activated objects and client-activated objects, the most explicit and therefore safest way for versioning remoteable components is through interfaces. A new version of a remoteable object usually continues supporting the old interface, and just adding a new interface allows newer clients to use new functionality offered by the remoteable object.

When it comes to versioning of serializable objects, you have to define exactly the requirements for versioning in the future: will exchanging serializable objects between two different remoting objects be necessary without losing any new information introduced by newer versions of the serializable object? If yes, the very first implementation of the serializable object has to take this into consideration to not lose new information just because it doesn't understand it. But this will only happen if you cannot or are not allowed to update software based on an older version of your serializable object. Usually this happens when it comes to interaction between applications of two different departments within organizations or even more likely between two companies. In both cases, you don't have detailed control over what happens with the other application. But in this case, I think **Web Services might definitely be a better choice for doing application communication**—except you have some requirements for using .NET Remoting (e.g., if you rely on specific types of the runtime that cannot be transmitted through Web services, or if you rely on callback—in both cases I would really encourage you to rethink the architecture and try to find some more loosely coupled solutions for such scenarios).

Also, don't forget to use the `includeVersions="false"` option on sending formatters if a remote object (either client or server) requires receiving different versions of the serializable type. Otherwise, the runtime always tries to load the exact assembly version for the serialized object, and if it doesn't find it, it throws an exception. Remember that if you don't use includeVersions and install more than one version of the assembly in the Global Assembly Cache, the runtime definitely will find the right version of the assembly for deserialization. But because your application has potentially been built against a different version—different version means different type—in this case you will get an exception when assigning the deserialized instance to a variable of the wrong type. Therefore, on a machine that runs two clients with different versioning requirements for serialized objects, I'd suggest you avoid installing assemblies in Global Assembly Cache to steer clear of this problem. Last but not least, in general I'd avoid putting assemblies in GAC if possible and really only add them if it's really necessary.

One thing you should never forget is that versioning is not easy. It is not easy for a simple application, although it has become much easier with the .NET Framework because of its built-in support for versioning. And it is even harder for distributed applications, because outside of detailed technological aspects, you have to keep in mind all implications on operational management and organizational challenges, too. Therefore, again, keep versioning in mind from the very first moment in your software development process.

CHAPTER 9

.NET Remoting Tips and Best Practices

In Ithis chapter, I'd like to share some additional experiences that I've encountered in the past years using .NET Remoting. I will do this as a series of tips or "best practices." But first, I'd like to discuss some of the use cases and—maybe even more important—nonuse cases for .NET Remoting in your applications.

.NET Remoting Use Cases

In the previous eight chapters, you've gotten to know all features of .NET Remoting as you might encounter them in a typical distributed application. I have tried to write these chapters as objectively as possible without making any judgments on whether or not using a feature in a certain circumstance is a good idea.

First, let me assure you that every feature of .NET Remoting has a certain place.[1] If you're developing an application for 10,000 users in your corporation, however, you might have to choose a different feature set from the one you'd use when creating a remoting solution in which two Windows Forms applications communicate with each other on the same machine. With the former type of application, you might have take care to *avoid* using some of the .NET Remoting features as they would negatively affect scalability.

If you don't know me personally, you might be tempted to assume that I value .NET Remoting above all other means of developing distributed applications. You might also assume that I'll use .NET Remoting as a catch-all solution to any distributed application. I have to admit that you would be wrong. I'm a true believer of choosing the right tool for the given task. Beside .NET Remoting, you'll find a number of different technologies for the development of distributed applications, and each one has its particular use: Enterprise Services and COM+, direct TCP/IP socket connections, UDP datagrams, MSMQ messages, Web Services via HTTP, the Web Services Enhancements (WSE), SOAP messages via reliable infrastructures, SQL XML, and probably more.

1. Although I have to admit that the place for SoapSuds.exe seems to be quite limited these days.

But let's first look at some of the areas in which you can find .NET Remoting. The four main use cases in terms of remoting boundaries are as follows. It can be used if you want your method calls to

- Cross AppDomain boundaries.
- Cross process boundaries on a local machine.
- Cross a LAN.
- Cross a WAN or the Internet.

Cross-AppDomain Remoting

As soon as you create a new application domain in .NET, you are automatically using remoting behind the scenes to communicate between the two AppDomains. In this case, the remoting framework will set up all channels and sinks for you—and in fact it will use a highly optimized formatting process and an in-memory channel.

This provides for two different implications: a) you can't change formatters or add channel sink chains, and b) you don't have to care too much about it. It just works. You can use all .NET Remoting features without any problems.

In fact, cross-AppDomain calls are one of the primary use cases for .NET Remoting. They are so well integrated in the framework that you usually don't even notice you're using remoting.

Safe Features

- All

Cross-Process on a Single Machine

Let's assume you have two Windows Forms applications running on a single machine, and you want the two applications to be able to communicate with each other. Or suppose you have a Windows service that should exchange data with your GUI application. Which protocol can you use? Remoting!

This is one of the cases where the TCPChannel is extremely helpful as it allows you to specify `rejectRemoteRequests="true"` upon its construction, which limits all incoming connections to the ones originating from your own machine. No need to take too many precautions about security in this case. (However, if you use fixed port numbers for WinForms-to-WinForms communication, you might encounter trouble when running on Windows Terminal Services with two or more users trying to use your application at the same time.)

I have good news regarding the features of .NET Remoting—all of them work as expected on a local machine.

Safe Features

- All

Cross-Process on Multiple Machines in a LAN

OK, now we start to get real. This is your usual distributed LAN application. Applications of this kind can be separated into two additional categories:

- Single-server applications
- Applications that need to scale to multiple hosts in a network load balancing (NLB) cluster

Don't Use Events or Callbacks

No matter which category your application belongs to, I heavily recommend *not* using events, callbacks, or client-side sponsors for networked applications. Yes, it's possible to use them. Yes, they might work after applying one or another workaround. The real trouble is that they aren't exactly stable and don't really perform that nicely.

The reason for this stability/performance drawback lies in the invocation model. First, you have to make a decision on whether to invoke events synchronously or asynchronously from the server's point of view. In the first case, your server has to wait until all clients acknowledge (and process) the callback, which increases the request time by a magnitude. If, however, you decide to use them asynchronously, you might run into a number of different issues—of which ThreadPool starvation is only the smallest and lost events is the more critical.

But what if you need notification of clients? In this case, you should either look into User Datagram Protocol (UDP) or message queuing, depending on your need for reliability. The use of Microsoft Message Queue Server (MSMQ) allows your server-side application to send messages to listening clients without having to wait for acknowledgements of reception (or even wait for processing at the client). This allows for way better turnaround times for your requests.

I will cover asynchronous notifications in more depth later in this chapter in the section "Using Events and Sponsors."

How About Client-Activated Objects?

The main problem with CAOs is that they are always bound to the machine on which they have been created. This means that you can't use load balancing or failover clustering for these objects. If, on the other hand, you use SingleCall SAOs designed with clustering in mind, you could use Windows Network Load Balancing (NLB) quite easily to randomly dispatch the method invocations to one out of a number of available servers.

If you are running a single-server application, this doesn't matter too much for you. If, however, there is the slightest chance that the application has to scale out to a server-side cluster, than CAOs might limit its scalability.

But you shouldn't just be concerned about scalability: CAOs also affect you on a single server. When running SingleCall SAOs (and when strictly keeping all state information in a database), you can shut down and restart your server on demand. For example, you could upgrade to a newer version or apply some bug fix without having to tell any user to close and restart your client-side application. As soon as you use CAOs, however, you instantly lose this feature. If you restart a server in which you host CAOs, the client application will receive exceptions when calling any methods on these objects. CAOs aren't restored after restarting your server. Don't use them if you care about high availability and transparent failover, unless you want to write these features yourself.

I'll talk about clustering in more depth later in this chapter in the section "Scaling Out Remoting Solutions."

What's the Best Channel/Formatter?

I recommend the use of the HttpChannel with the binary formatter for any application that spans multiple hosts. The reason is quite simple: you can develop and debug your application in a standard Visual Studio .NET project, but when it comes to deployment, you can easily host your server-side components in IIS, which provides you with a number of advantages—a reasonable process model, built-in authentication (HTTP Basic or Windows integrated), built-in encryption (SSL), and the ability to disable HTTP KeepAlives, which further increases the scalability of your application because it reduces dependencies on single servers.

Safe Features

- SingleCall SAOs hosted in IIS with HttpChannel and BinaryFormatter

That's it. If you want to be on the safe side, don't use more than these features. Also, please keep in mind that whenever you return a MarshalByRefObject from a server-side method, you are actually creating an object that behaves like a CAO and should therefore be avoided.

Cross-Process via WAN/Internet

As soon as your application grows and you leave the boundaries of your local area network, a number of additional issues have to be taken care of. The absolute number one issue is network latency. You have to take care to reduce the number of cross-network calls by using chunky interfaces. In a chunky interface, you will try to transfer as much data as possible (and necessary) in a single network roundtrip. If your client application, for example, works with customer objects and addresses, then you should definitely transfer all known addresses for a given customer whenever the client application requests information about a customer. The alternative of having two different methods, `GetCustomer()` and `GetAddresses()`, will simply double the number of network roundtrips and will therefore heavily decrease your application's response times.

But keep in mind that you have to strike a balance here. It might not be the best idea to transfer all of the customer's orders or the complete contact history at the same time, if you don't need that data in 99 percent of the cases. It's really all about balance here.

I guess the most important advice I can give you for applications like this is to actually develop them with a low-bandwidth, high-latency network. Run your server and client on different machines not connected by a LAN. Instead, connect the client to the Internet with a plain old modem or ISDN line, if this is what you expect your users to use. This will allow you to see and experience the performance hot spots during development—nothing is more embarrassing than having a user call you up and tell you that your application is way too slow, right?

Regarding the use of remoting features, I can give you basically the same advice as for the LAN environment: use SingleCall SAOs hosted in IIS with HttpChannel and BinaryFormatter. In addition, you have to make absolutely sure that you don't use events, callbacks, or client-side sponsors, as these might not work whenever a firewall, proxy, or NAT device is used between the client computer and your server. Whereas the use of events in a LAN environment might "just" render your application instable, they will simply prevent it from working in WAN environments.

When using .NET Remoting on a WAN or the Internet, it is very important to develop with a sound versioning strategy. In practice, this for example means that you shouldn't use [Serializable] without implementing ISerializable. The reason is that you usually cannot force all worldwide users who run your application to install a new version right at the instant it becomes available.

You will usually have to support multiple versions of your client software at the same time. Even if the WAN is completely controlled by your own organization, you will still have a hard time convincing your traveling salespeople to download a new 5- to 10-megabyte installer if they only connect via GPRS from their cell phones.

Whenever you use .NET Remoting on the Internet, however, you have to be aware of the issue with HTTP proxying, as discussed in the section "Authenticating HTTP Proxying."

Safe Features

- SingleCall SAOs hosted in IIS with HttpChannel and BinaryFormatter

Nonusage Scenarios

After presenting four different application scenarios for remoting in the previous sections, I'd also like to point out some environments in which I wouldn't use remoting at all.

Let's Do SOAP

If you plan on using SOAP Web Services to integrate different platforms or different companies, I really urge you to look into ASP.NET Web Services (ASMX) instead of remoting. These Web Services are built on top of industry standards, and together with additional frameworks like the Web Services Enhancements, will allow you to use additional infrastructure-level and application-level specifications (WS-Security, WS-Routing, WS-Policy, WS-Trust, WS-SecureConversation, etc.) in a platform-independent and message-oriented way.

ASMX Web Services provide essential features for Web Services like WSDL-first development, the use of doc/literal, easier checking of SOAP headers, and so on.

So let me repeat: if you want SOAP, the use of ASP.NET Web Services together with WSE is the only way to go!

Service-Oriented Architectures

One of the current industry buzzwords is *Service-Oriented Architecture* (SOA), which provides platform-independent, message-oriented, and loosely coupled services in Enterprise environments. As you might already have guessed, remoting is not the right choice for these. Think about going ASMX and WSE here as well.

The reason is that remoting depends on the .NET runtime *and* on the availability of a binary interface contract (in the form of the "real" server-side classes or in the form of .NET interfaces) on the client machine. These are two requirements that essentially bind this framework to .NET.

Authenticated HTTP Proxying

.NET Remoting generally supports HTTP proxies with both the SoapFormatter and the BinaryFormatter. Unfortunately, it isn't possible to use proxies that require any form of authentication before allowing to send any data to remote hosts.

You can, however, look into the third-party product Genuine Channels,[2] which provides an HTTP channel with support for authenticated proxy traffic.

---

2. Genuine Channels is an independently developed product. The author has no relationship with this company. You can reach them at `http://www.genuinechannels.com`.

Distributed Transactions, Fine-Grained Security Requirements, Etc.

A completely different no-go scenario for remoting is the necessity for distributed transactions, fine-grained security requirements, configurable process isolation, publish-and-subscribe events, and so on. Yes, you could in fact develop your own channel sinks and plug them into the .NET Remoting framework to enable these features. But why would you want to do so? Why should you spend your time on these features instead of implementing your application's business requirements? There already is another framework in .NET that includes all these features: Enterprise Services.

If your application can make use of any of the following services, you should really think about using Enterprise Services instead of .NET Remoting:

- Distributed declarative transactions
- Highly flexible, configurable means of authentication and authorization
- Role-based security with roles independent of Windows user accounts
- Just-in-time activation of objects
- Object pooling
- Process isolation
- Server-side components as Windows Services
- Automatic queuing of component interactions with MSMQ

In addition, COM+ 1.5, as it is available with Windows Server 2003, provides the added benefit of so-called Services Without Components (SWC). This allows you to use most of the services of the Enterprise Services framework without the necessity to derive your component from System.EnterpriseServices.ServicedComponents and without having to register your components in the COM+ catalog.

To learn more about Enterprise Services and COM+, I'd like to point you to Juval Löwy's book, *COM and .NET Component Services* (O'Reilly, 2001).

The Nine Rules of Scalable Remoting

As you've seen previously, remoting provides a number of features whose applicability differs for a number of usage scenarios. To ensure that your application is reliable, stable, and scalable, you should follow these nine rules if you want to create highly scalable remoting solutions:

- Use only server-activated objects configured as SingleCall.
- Use the HttpChannel with the BinaryFormatter. Host your components in IIS if you need scalability, authentication, and authorization features.
- Use IIS's ability to deactivate HTTP KeepAlives for maximum scalability.
- Use a Network Load Balancing cluster of servers during development if you want to achieve scalability. Make sure to deactivate any client affinity and make sure that you deactivate HTTP KeepAlives during development!

- Don't use client-activated objects, and don't pass any MarshalByRefObject over a remoting boundary. Starting with version 1.1 of the .NET Framework, a SecurityException or a SerializationException would be thrown in this case. (Yes, you could change the underlying TypeFilterLevel setting—but you shouldn't!)
- Don't use events, callbacks, and client-side sponsors.
- Use static fields and other forms of caching with caution. If you keep state in memory, you might run into problems if you try to scale your application out to a cluster of servers. Cache information only if it's not going to change or if you can anticipate the level and number of changes in advance. Otherwise, you will run into cache-synchronization nightmares on your cluster.
- Don't use remoting for anything else apart from .NET-to-.NET communications. Use ASP.NET Web Services and WSE for anything related to SOAP, Service-Oriented Architectures, and platform independence.
- Don't try to fit distributed transactions, security, and such into custom channel sinks. Instead, use Enterprise Services if applicable in your environment. .NET Remoting isn't a middleware, it is just a transport protocol—if you need services, use a service-oriented framework!

Please keep in mind that these guidelines are only for remoting solutions that should scale out. If you communicate between two applications on a single server, it is usually perfectly sound to use some of the more advanced features of .NET Remoting.

Using Events and Sponsors

When looking at the remoting of events for the first time, it seems like a perfect solution for a recurring design question: what if I need to notify a number of clients of important changes in data? Can't they just subscribe to a certain server-side event that will be invoked later?

The theoretical answer is yes. In practice, however, the way .NET Remoting handles these kinds of events might not be up to your requirements of stability and reliability. Actually, it's not really all the fault of the .NET Remoting framework, but is partly caused by the way the TCP protocol works. But let's start with the basics: let's assume that you have a few hundred clients that subscribe to a server-side event. From the previous chapters, you already know that whenever an event is invoked, a TCP connection originating from the server to the client is established. The invocation of the event is transferred to the client as if it were a normal .NET Remoting method call. In fact, it is just this: a normal method call with the main distinction that client and server have changed their roles.

Knowing this, you now have basically two options: you can notify all clients sequentially or in parallel. If you want to notify them in sequence, you have to take into account that the distribution of data might take a while, as some clients might react more slowly than others. In fact, if a client couldn't respond in time, then this might in turn cause your server application to hang.

However, if you decide to notify all clients in parallel—for example, by using `BeginInvoke()`—you can run into another issue. The .NET Remoting framework uses the underlying standard .NET thread pool to handle incoming requests. This pool consists of 25 threads per CPU.

Unfortunately, the same thread pool is employed whenever you use `BeginInvoke()`. This means that your server will not be able to handle incoming requests while it is processing the outgoing events 25 at a time.

As you can see, the support for events in the .NET Remoting framework has not been designed for global broadcast operations. Listeners should be confined to applications running on the same machine, for example, in communication with a local Windows service that uses .NET Remoting as its preferred protocol.

How to Notify Nevertheless

But what if you nevertheless have to notify a number of clients? I am afraid that .NET Remoting might not be the best solution for the notification scenario (although you can, of course, still use it for all client-to-server communication.) Instead, you should take a look at technologies like UDP, MSMQ, or IP Multicasting, which allow you to broadcast events efficiently to a number of subscribers.

UDP broadcasts can be used if all clients are located in the same IP subnet and if you don't need reliable delivery (i.e., if the events are not critical and when it wouldn't matter whether a small percentage of them are never received at all clients). In addition, UDP limits the payload to 64KB. The main advantage of using UDP broadcasts is that only a single IP packet will be sent to your network, no matter how many applications are listening.

A simple UDP client that listens for broadcast packets on port 10000 will look like this:

```
using System;
using System.Text;
using System.Net.Sockets;
using System.Net;

public class Receiver
{
  public static void Main()
  {
    Socket sck = new Socket(
      AddressFamily.InterNetwork,
      SocketType.Dgram,
      ProtocolType.Udp);

    sck.Bind(new IPEndPoint( IPAddress.Any, 10000));

    byte[] buf = new byte[65000];

    while (true)
    {
      int size = sck.Receive(buf);
      String str = Encoding.ASCII.GetString(buf,0,size);
      Console.WriteLine("Received: {0}", str);
    }
  }
}
```

The following application is a matching sender that allows the user to enter a string that should be broadcast:

```
using System;
using System.Text;
using System.Net;
using System.Net.Sockets;

public class Sender
{
  public static void Main()
  {
    Console.Write("Enter String to broadcast:");
    String str = Console.ReadLine();
    byte[] data = Encoding.ASCII.GetBytes(str);

    Socket sck = new Socket(
      AddressFamily.InterNetwork,
      SocketType.Dgram,
      ProtocolType.Udp);

    sck.Connect(new IPEndPoint( IPAddress.Broadcast,10000));
    sck.Send(data);
    sck.Close();
  }
}
```

When using broadcasting schemes based on UDP, you have to be aware that events might be lost and that a single packet might be received multiple times if the code binds the socket to all existing network interfaces. This has been done in the preceding code by using IPAddress.Any at the receiving side, and by using IPAddress.Broadcast on the sending side.[3]

Message Queuing to the Rescue

Contrary to UDP, MSMQ is the technology of choice if you need guaranteed asynchronous delivery of notifications.

■ **Note** MSMQ is a built-in part of Windows XP, Windows 2000, and Windows Server 2003 that has to be selected during installation and might not be available on all client machines by default. There are no additional licensing costs for using MSMQ in your applications.

---

3. If you are interested in more details, I'd recommend having a look at the book *Pro .NET 1.1 Network Programming, Second Edition* by Christian Nagel et al. (Apress, 2004).

By default, MSMQ does not use broadcasting or multicasting, but instead requires that the data for each client be sent in a separate connection. This sending is, however, performed in the background by the MSMQ Server running on the sending machine. (Machines using the current version of MSMQ are server and client at the same time!) The sending application basically only needs to tell the messaging framework to take care of delivering the messages but does not have to wait for completion.

Whenever you want to use MSMQ for notifications, you will usually create a destination queue on each of your clients. This can be done programmatically during installation or use of your application (if the user has the necessary permissions to create queues).

A very simple receiving application can, for example, look like this (you have to add a reference to System.Messaging.DLL):

```
using System;
using System.Messaging;

class Receiver
{
  static void Main(string[] args)
  {
    String queuename = @".\private$\NOTIFICATIONS";

    if (!MessageQueue.Exists(queuename))
    {
      MessageQueue.Create(queuename);
    }

    MessageQueue que = new MessageQueue(queuename);
    que.Formatter = new BinaryMessageFormatter();

    while (true)
    {
      using (Message msg = que.Receive())
      {
        String str = (String) msg.Body;
        Console.WriteLine("Received: {0}", str);
      }
    }
  }
}
```

This application creates an incoming private queue called "NOTIFICATIONS" and waits for incoming messages.

■ **Note** "Private" queues can be reached from other machines if the sender knows their exact name. They simply do not appear in the Active Directory (AD).

A matching sender application that can send notifications to multiple receivers could, for example, look like this:

```
using System;
using System.Text;
using System.Collections;
using System.Messaging;

class Sender
{
  static void Main(string[] args)
  {
    Console.Write("Enter String to broadcast:");
    String str = Console.ReadLine();

    ArrayList clients = new ArrayList();
    clients.Add("localhost");
    clients.Add("client1");
    clients.Add("client2");

    String formatName = BuildFormatName(clients);
    MessageQueue que = new MessageQueue(formatName);

    Message msg = new Message();
    msg.Formatter = new BinaryMessageFormatter();
    msg.Body = str;
    que.Send(msg);

    Console.ReadLine();
  }

  static string BuildFormatName(ArrayList clients)
  {
    if (clients.Count == 0)
      throw new ArgumentException("List of clients empty.", "clients");

    StringBuilder bld = new StringBuilder();
    bld.Append("FormatName:");
    foreach (String cli in clients)
    {
      bld.Append("direct=os:");
      bld.Append(cli);
      bld.Append("\\private$\\NOTIFICATIONS");
      bld.Append(",");
    }
    bld.Remove(bld.Length-1,1);
    return bld.ToString();
  }
}
```

In this example, the method `BuildFormatName()` takes a list of clients that should receive the notification and builds a destination name containing all the individual queues.

If you don't want to hard code the queue names on the client side (for example, if multiple instances of your application can be started on one host), it is advisable to create a new queue upon application startup, giving it a random or GUID-based name. You would then contact the server (by either using MSMQ or maybe also using .NET Remoting) to have it add the newly created queue to its list of subscribers. You would then also have to include mechanisms for unsubscribing, deleting the dynamically created queues, and detecting stale entries in the list of subscribers.

Other Approaches

Delivering notifications to a large number of clients is a very complex topic for which a number of additional strategies have been developed depending on the usage scenario. An approach not discussed previously is, for example, the use of TCP connections that have been created by a client solely for the purpose of allowing a server to use it for notifications. Point-to-point UDP connections and HTTP-based "polling" are other approaches that might be necessary depending on the number and kinds of clients your application should support.

SoapSuds vs. Interfaces in .NET Remoting

Whenever people have approached me in the previous years to ask for my opinion on SoapSuds, I have been recommending using interfaces to access remote objects instead. My exact words might even have been close to *"I'd generally recommend avoiding SoapSuds whenever possible."*

The idea behind SoapSuds is to run it on an existing assembly to extract the metadata for all MarshalByRefObjects so that you don't need to deploy the complete implementation assembly to your clients. This goal is pretty ambitious, and in fact, I truly believe that it just can't work. As soon as you have [Serializable] or ISerializable classes in your assembly, you are pretty much on your own because SoapSuds will only extract the metadata (i.e., the fields) but not the implementation. Even if it would extract the implementation, this might also not match your expectation because it might contain source code that should only run on the server side.

Just imagine that you have a class like this:

```
[Serializable]
public class Foo
{
   private String _bar;

   public String Bar
   {
     get { return _bar; }

     set
     {
        if (value.Length > 30)
        {
            throw new ApplicationException(
                 "Bar might not be longer than 30 chars");
```

```
        }
        _bar = value;
     }
  }
}
```

When running SoapSuds here, it will extract only the metadata, which basically leaves you with the following class, which is quite a bit different:

```
[Serializable]
public class Foo
{
   public String Bar;
}
```

But even if you can work around this issue (by including your [Serializable] classes in a different assembly that is shared between server and client), you might still run into some issues.

As soon as your application increases in complexity, you encounter one or more of the following problems, depending on the version of the .NET Framework and its service packs:

- Typed DataSets are not supported by SoapSuds.
- If you use System.ComponentModel.Component (and some others), SoapSuds will simply throw an exception instead of generating anything.
- Various conditions trigger the generation of non-compilable code (duplicate using statements in a file, and so on).
- Async calls via `Delegate.BeginInvoke()` won't work.

One of the reasons for using SoapSuds is the ability to register these metadata-only classes at the client side so that you can basically use the `new` operator to instantiate remote references. But at the end of the day, location transparency—as it is implied in this case—might even turn out to be dangerous for an application's stability. It can affect your application's performance in negative ways (for example, when using way too chatty interfaces). Normally, you should know exactly which method will be executed remotely and which class will run in a remote context—therefore my conclusion:

- Use explicitly defined remote interfaces.
- Use a helper class like the one shown here if you want to go with configuration files.
- Use factory SAOs instead of CAOs ("activated" types don't work with interfaces, therefore, factory). Or even better: avoid CAOs if possible, especially if your application should support transparent failover on a cluster.

Let me give you one more reason why I definitely advocate the use of explicit interfaces to access remote objects. Just imagine that you inherit client-side code written by someone else and you see code like the following.

```
private double CalculateSum(Order o)
{
    double sum = 0;
    foreach (OrderDetail od in o.Details)
```

```
    {
         sum = sum + od.LineTotal;
     }
     return sum;
}
```

Your user complains that the "application is too slow," but you just can't find any problems. You didn't instantly notice that Order and OrderDetail are actually configured as remote objects and a single execution of this method might result in a dozen or more network roundtrips.

If you would have used interfaces instead, problems like this can be much more obvious.

```
private double CalculateSum(IRemoteOrder o)
{
    double sum = 0;
    foreach (IRemoteOrderDetail od in o.Details)
    {
         sum = sum + od.LineTotal;
     }
     return sum;
}
```

Actually, if the initial developer would have used interfaces right from the start, the code might have looked differently. As soon as he or she noticed that this method would involve multiple roundtrips, he or she might have changed the interface and calculated the complete value at the server.

```
private double CalculateSum(IRemoteOrder o)
{
    return o.CalculateSum();
}
```

That's why I use interfaces. They help me to avoid mistakes.

Custom Exceptions

Even though remoting supports high type fidelity, transferring custom exceptions over remoting boundaries comes with its very own challenge: using [Serializable] is not enough because the base class System.Exception already implements ISerializable. If you want to pass your custom exceptions over remoting boundaries, you therefore have to override `GetObjectData()` and provide a custom constructor for deserialization. And don't forget to call `base.GetObjectData()`.

Please also note that your custom exception has to be deployed to a shared DLL that is copied to, and referenced from, your client and server applications. It doesn't suffice to simply copy and paste the source code to your client and server project!

Any custom exception you develop should be based on the following skeleton:

```
[Serializable]
public class MyException: ApplicationException
```

```
{
  public MyException(): base()
  {
  }

  public MyException(String msg): base(msg) {}

  public MyException(SerializationInfo info,
      StreamingContext context): base(info, context)
  {
  }

  public override void GetObjectData(SerializationInfo info,
      StreamingContext context)
  {
    base.GetObjectData (info, context);
  }
}
```

If you want to transfer custom information with your exception, you have to add it to the SerializationInfo object on the call to `GetObjectData()` and to take it from this object in the secondary constructor.

```
[Serializable]
public class ConcurrencyException: ApplicationException
{
  string _databaseTable;

  public ConcurrencyException(): base()
  {
  }

  public ConcurrencyException(String msg, String databaseTable): base(msg)
  {
    _databaseTable = databaseTable;
  }

  public ConcurrencyException(SerializationInfo info,
      StreamingContext context): base(info, context)
  {
    _databaseTable = info.GetString("table");
  }

  public override void GetObjectData(SerializationInfo info,
     StreamingContext context)
  {
    base.GetObjectData (info, context);
    info.AddValue("table", _databaseTable);
  }
```

```
  public String DatabaseTable
  {
    get { return _databaseTable; }
  }

}
```

If you don't override `GetObjectData()` and/or don't provide the additional constructor `ConcurrencyException(info, context)`, you would instead end up with a very different exception in your Catch block:

```
The constructor to deserialize an object of type General.ConcurrencyException was
not found.
```

This exception simply indicates that the client-side of the remoting framework has not been able to correctly deserialize your custom exception (called ConcurrencyException in this sample).

Scaling Out Remoting Solutions

As .NET Remoting is a TCP/IP-based RPC mechanism, you can scale out remoting solutions onto a cluster using standard Windows Network Load Balancing (NLB) without the need to purchase and implement additional software.[4]

Load Balancing Basics

Network Load Balancing is an easy way to distribute incoming requests amongst a number of worker servers. Windows NLB—as it is built into Windows 2000 Server and Windows Server 2003—is a load-balancing implementation that works for up to 32 nodes without the necessity to define a dedicated "gatekeeper" machine. This means that you can take advantage of load balancing with as little as two standard server machines.

To illustrate the behavior of a cluster, let's assume that you have two servers, SERVER01 and SERVER02, with the following IP addresses:

- `SERVER01: 192.168.0.41`
- `SERVER02: 192.168.0.42`

Using Windows NLB, you can now create a cluster with a so-called virtual IP Address (VIP) of, for example, 192.168.0.40, resulting in the configuration shown in Figure 9-1.

4. It is, of course, also possible to run your remoting solution using Application Center 2000 or third-party TCP/IP load-balancing appliances, but this is beyond the scope of this book.

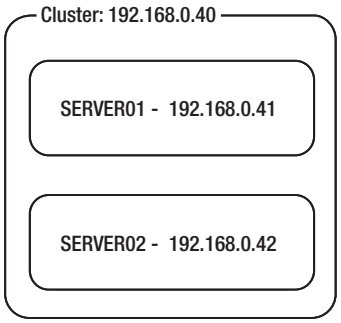

Figure 9-1. *A cluster's IP addresses*

To leverage the load-balancing capabilities of Windows, you will now have to deploy the exact same remoting services on both machines. You can, for example, copy the same server-side configuration files and bin\ subdirectory to an IIS virtual directory on both machines.

Instead of accessing this service using one of the two servers' real IP address, you will then configure the clients to use the cluster's virtual IP address as the server's location. You can, for example, access a correctly configured service on the cluster presented earlier using the following line of code:

```
ICustomerManager mgr = (ICustomerManager) Activator.GetObject(
        typeof(ICustomerManager),
        "http://192.168.0.40/Remoting/CustomerManager.rem");
```

Whenever a client establishes a TCP/IP connection to the cluster's virtual IP address, Windows NLB selects one of the cluster's nodes to handle the connection request.

■ **Note** There are essentially two different kinds of load-balancing infrastructures. One is based on IP-level load-balancing devices that sit right in front of your network (from the client's point of view) and that distribute the load among the nodes in a cluster. In the second way, which is implemented by Windows Network Load Balancing, however, there is no such need for an additional device. Request distribution is instead handled by a distributed algorithm running on all nodes in a cluster.

NLB for Throughput and High Availability

Using Network Load Balancing as illustrated previously provides a great way to increase your application's throughput (if you reach CPU limits) and availability. Most people unfortunately look at NLB only if they need the additional throughput, which can be achieved by scaling out a solution onto a cluster of machines. However, I personally recommend the use of NLB clusters even for smaller setups in which the performance requirements could easily be fulfilled with a single server.

The reason for this is simple: increased availability. If you design your application correctly, then it is possible to take nodes offline from your cluster without affecting a single running client. All future requests will then be handled by the remaining node(s). This allows you to install new versions of your application or service packs or even update operating systems without taking your application offline.

After you've finished administrative work on one node, you allow it to join the cluster again; you can then proceed to take the next node offline.

■ **Note** Windows NLB is compatible across all supported operating systems. You could have some nodes running on Windows NT 4.0, and a few on Windows 2000, while at the same time updating the remaining part of your cluster to Windows Server 2003. As I personally believe that this will also be true for future operating systems, it essentially means that you will never again have to take your application offline for scheduled maintenance—even if you upgrade your operating system. Your system administrators will love you for not having to work during late night and weekend maintenance timeslots.

HTTP, TCP, Connections, and Sessions

I am sure that you are already looking forward to getting your hands dirty with your first NLB cluster configuration. But before I show you how to set up an NLB cluster (which shouldn't take much more than about 15 clicks), there is one more important technical peculiarity that I have to tell you: as briefly mentioned earlier, NLB actually only balances connection *requests*. As soon as a TCP connection has been established, all future communication using the specific socket link will take place between the client and the originally selected cluster node. If a single cluster node fails, all clients that currently have open communication links to it will therefore receive an exception, telling them that the connection has been dropped. This wouldn't be that big of a problem if it weren't for .NET Remoting reusing the underlying TCP connection.

Remoting uses the so-called HTTP 1.1 KeepAlive functionality. This means that multiple HTTP requests will be sent using a single TCP/IP connection. The connection will be kept open for a certain period of time (usually around two minutes), and all further requests occurring during that time window will be sent over the same connection. This is usually a good thing because it eliminates the need for reestablishing dozens of TCP connections whenever you open a conventional Web page containing multiple images. Otherwise, the client would have to open separate TCP connections for each and every image file.

In cases when you want to provide maximum availability for your application, this Web-originating optimization in the HTTP protocol actually causes a number of problems. As soon as a node fails, all clients who currently have an open (cached) connection to this node will receive exceptions whenever they invoke a method on the server-side object. In this case, it would actually be beneficial if no connections were cached at all so that every single remoting request causes a new TCP connection that could then be load balanced on its own.

Fortunately, this is easily possible when hosting your remoting components in IIS—which is actually the only load-balancing configuration that is officially supported by Microsoft. You just have to open an IIS management console and navigate to the Web site that contains your remoting servers. After right-clicking, you will be presented with the dialog box shown in Figure 9-2 in which you can uncheck the checkbox option "HTTP Keep-Alives Enabled."

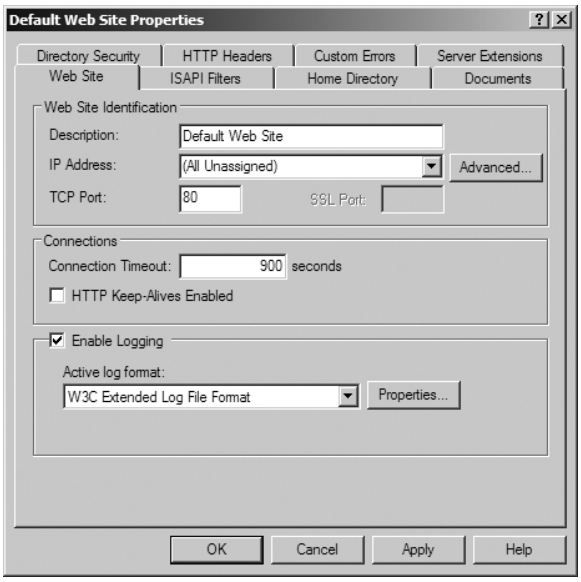

Figure 9-2. *Disabling HTTP KeepAlives*

Please note that you should not run any "real" Web sites on the same virtual roots, as their performance usually depends on this setting. For remoting applications, the only drawback on a LAN environment is a small performance degradation of around 2 msecs per remote procedure call. Given the gained advantage of scalability, higher throughput, and higher availability, this is usually by a magnitude worth it.

■ **Note** This is also the reason why all scalability features are currently only supported when using the HTTP channel hosted in IIS. Neither the TcpChannel nor the HttpChannel, when hosted in a custom application, allow you to disable these connection caches.

Creating Your Cluster

In the following section, I'll work with you step by step on creating a cluster that you can use to develop scalable systems. I do this primarily for one reason: these applications cannot be developed or tested without one. It is nearly impossible to successfully create an application on a single machine and deploy it on a cluster without any issues. That's why I generally recommend doing all your development tests on a cluster if you plan to deploy on one.

■ **Caution** Before showing you how easy it is to create an NLB cluster, let me give you some words of caution. There are a number of options and trade-offs that have to be made when creating NLB clusters. I would recommend that you look in the documentation of this feature, which contains a detailed checklist of things to consider before creating a production-level cluster. If in doubt, please contact an experienced system administrator, as the functionality of an NLB cluster not only depends on your computer hardware, but might also—in a few instances—be affected by your network hardware or layout. Your system or network administrators will usually be able to help you.

I will in addition assume a certain level of knowledge about setting up IP networks using the Windows operating system. And as a fair bit of warning: if you run in a corporate environment, please check with your system and network administrators whether it's OK to do the following. Administrators of large networks tend to get somewhat mad at people who pick and choose their own IP addresses and similar.

Let's Get Started

The cluster you are about to build runs with two PCs, each containing a single network interface card (NIC). It can be built out of any off-the-shelf standard hardware; for demonstration purposes, you can even build a cluster using a couple of laptop computers running Windows Server 2003.

As in the earlier example, I assume that your servers are currently configured for the IP addresses 192.168.0.41 and 192.168.0.42, and that your future cluster's virtual IP address should be 192.168.0.40. The first step in building your cluster is to go to a machine running Windows Server 2003 and navigate to Start ➤ Administrative Tools ➤ Network Load Balancing Manager. The tool doesn't have to be run on one of the future cluster's node. NLB Manager opens with the user interface shown in Figure 9-3.

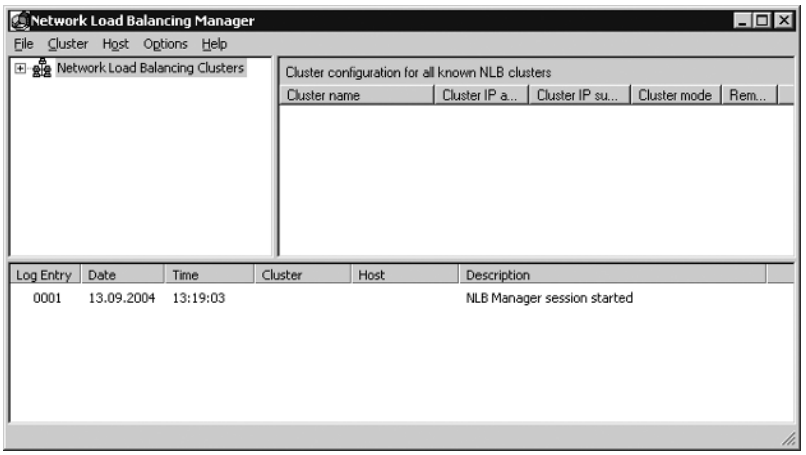

Figure 9-3. *Starting the NLB manager*

To create a new cluster, please navigate to Cluster ➤ New. In the dialog box that is opened (shown in Figure 9-4), you have to specify the cluster's future IP address, its subnet mask, and domain name. In addition, you have to select an operation mode. Describing the subtle differences between these two modes is definitely beyond the scope of this section, but let me simply rephrase the documentation: use Multicast if possible. If your network infrastructure does not support it, fall back to Unicast.[5]

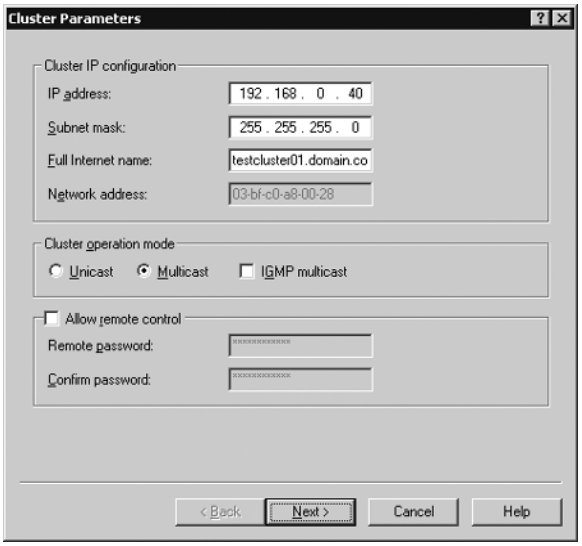

Figure 9-4. *The cluster's basic information*

After you click the Next button, you will be presented with an option to specify additional virtual IP addresses. For most clusters, a single IP address should be enough, so that you can skip this step by clicking Next.

As you can see in the following dialog box, which is shown in Figure 9-5, the process is getting more interesting. It allows you to specify "port rules" that define the policy (or policies) that will be used to determine which nodes in a cluster will be asked to handle a certain connection request.

5. I would seriously suggest that you refer to the online documentation for NLB to fully understand the implications and limits of using Unicast load balancing.

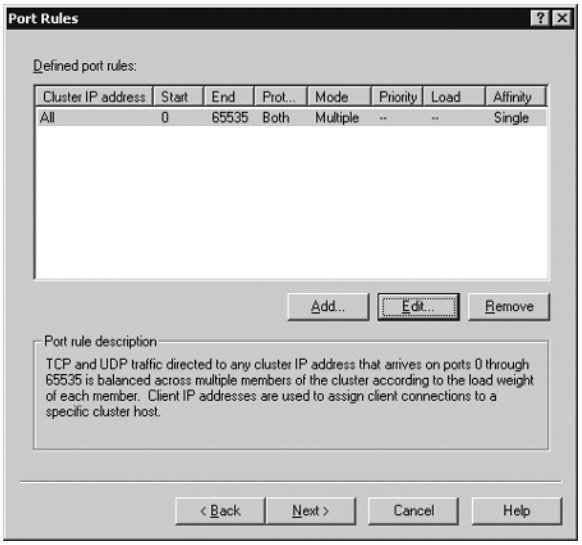

Figure 9-5. *Specifying port rules for your cluster*

If you double-click the default rule that has been created for you, you will get a dialog box that allows you to define port ranges which govern the load distribution. As you can see in Figure 9-6, I've made a small adjustment to the session affinity setting by changing it from Single to None.

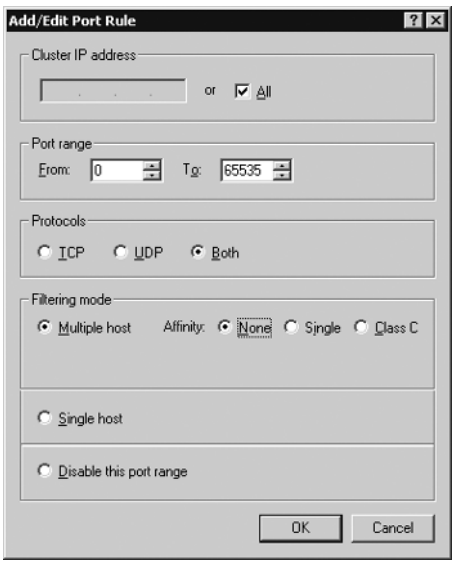

Figure 9-6. *Disabling client/server affinity*

This setting determines whether or not subsequent incoming requests originating from a same single IP address or Class C IP subnet will always be routed to the same node in a cluster. This is, for example, necessary if you run a Web site that uses the old ASP Session model of pre-.NET times. In this case, the session state has been stored in memory and is therefore tied to a single machine. Using session affinity allows the developer to be relatively sure that all incoming requests from a client will always be forwarded to the same machine that contains the session data.

■ **Note** This has never been a reliable setting because some—especially larger—providers employ transparent HTTP proxy arrays, which can cause subsequent requests from a single client to be tunneled via different HTTP proxies (maybe even in different class C subnets). From a server's or cluster's point of view, each request would therefore be originating from a different client without any kind of session affinity. As an alternative, there are third-party network appliances that can, for example, look at an HTTP cookie instead to determine the correct host to handle an incoming session. But this is mostly only relevant for Web sites but not for application servers.

After clicking OK and confirming the underlying dialog box, you can proceed to add nodes to your cluster. To do so, you need to specify a future node's IP address, click Connect, select a LAN interface on the node, and click Next as illustrated in Figure 9-7.

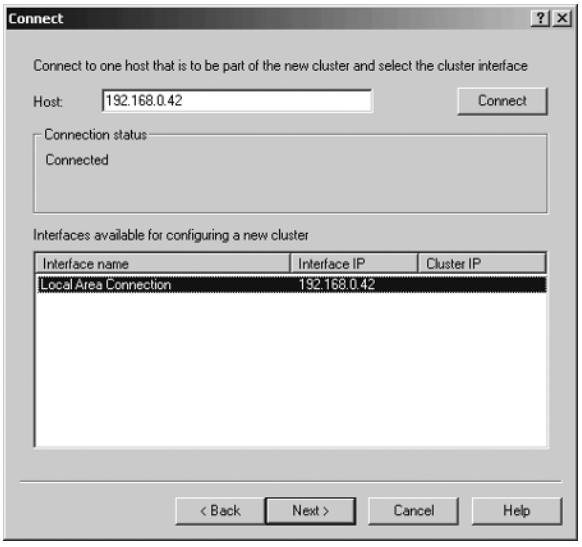

Figure 9-7. *Adding a node to a cluster*

The final step before a node is added to a cluster is to specify a unique host ID and confirm the remaining settings as shown in Figure 9-8.

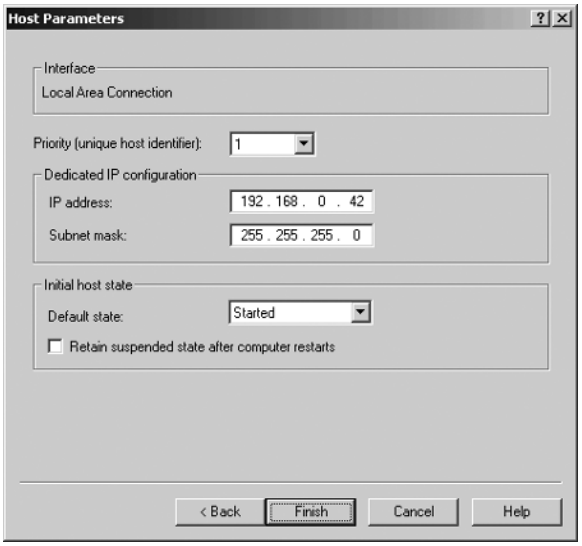

Figure 9-8. *Node details*

As soon as you finish this dialog box, the cluster will be built. The Network Load Balancing Manager will use Windows Management Instrumentation (WMI) to connect to the specified nodes to configure the cluster settings. Please note that it might take a few seconds or minutes for these changes to be performed.

After the first node has been added, you will be presented with the cluster overview as shown in Figure 9-9. Here you can right-click the cluster node and choose Add Host To Cluster to add the second node by performing the last two steps again for a different IP address.

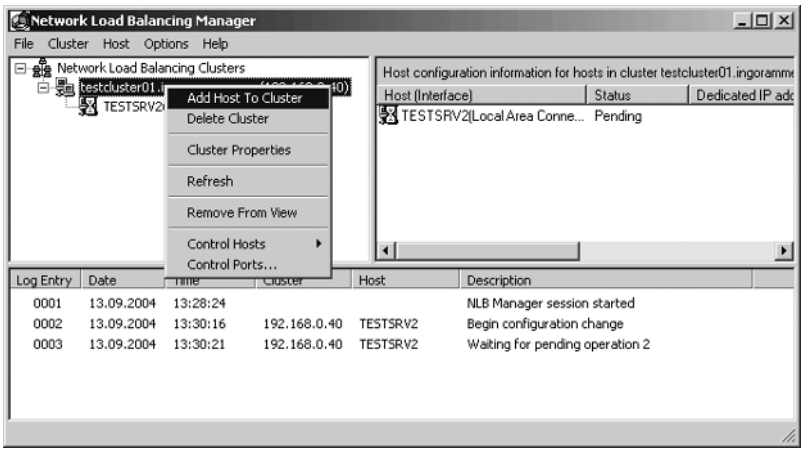

Figure 9-9. *Overview of your new cluster*

As soon as the configuration process has been completed, you can send requests to the cluster's virtual IP address.

Taking Nodes Online/Offline

I have previously told you about the fact that using NLB clusters to host your remoting components allows you to update your running server systems without affecting any client. Simply unplugging a node from the network (which would in fact simulate the failure of a node) would result in the currently executing requests being cancelled.

To reduce the effect on your clients to an absolute minimum, you can instead "drain" the connections. To do this, right-click a cluster host in NLB Manager and select Control Hosts ➤ Drainstop. As soon as you choose this option, the selected node in the cluster will not accept further connection requests, but will still complete all currently running interactions. Once all connections have been closed, the node will automatically be temporarily removed from the cluster.

After stopping a node, you can essentially apply any kinds of updates and system changes as the selected node will not respond to incoming requests. As soon as your administrative work has been completed, you can allow the node to rejoin the cluster by clicking Control Hosts ➤ Start in NLB Manager.

Designing Applications for Static Scalability

To ensure that your application runs correctly in a cluster, you have to take a number of precautions during design and development. If you don't watch out for the following, then your application will appear to run correctly but might produce incorrect results.

The most important design and development consideration for scalable applications is to take special care about all in-memory state of your application. More formally speaking, you have to make a distinction between shared storage and local storage. Shared storage, like a database, is used by multiple nodes at the same time so that each node can reasonably assume that the content is correct and current. However, if you keep in-memory copies of your data at each node, then each modification is processed independently so that the node-specific caches can divert, resulting in the situation that each node has a different value in its cache.

But let's look at an example. Let's assume that you've designed a system that manages customer information using an interface containing the following two methods:

```
interface ICustomerManager
{
  Customer GetCustomer();
  void StoreCustomer(Customer cust);
}
```

To increase performance, you decide that customer information wouldn't necessarily need to be read from the database. Instead, you simply cache this information in memory. Whenever `StoreCustomer()` is called, you first update the database and, if successful, afterwards also update the server's in-memory cache.

This might seem like a reasonable way to approach this problem if your application only runs on a single server and if nobody else directly accesses your database. As soon as you scale out onto a cluster, this application will, however, produce incorrect results. Let's assume that the name of customer #123 is originally "John Doe". As soon as a request to change this name

reaches one of the nodes, it will update the database and its in-memory cache to reflect the new value of, say, "John Moe". If a second request to retrieve the customer information for customer #123 hits a different node, the returned data will, however, still contain the old—and meanwhile incorrect—value of "John Doe".

Does this mean that you cannot use any in-memory caches when planning to deploy your application in a cluster? No, definitely not. It just means that you have to look quite carefully at the type of data you are working with.

Therefore, one of the first steps in creating scalable applications is to analyze the different kinds of data that is used by your application. It can be advantageous to split the list of database tables into three groups:

- *Group 1*: Static data.
- *Group 2*: Near-static, seldom changed data. Data that is updated in advance or that is only changed during certain time windows (for example, daily at midnight).
- *Group 3*: Operative, dynamic data.

If you look at an online shop as a sample, static data are usually lists of zip codes, cities, payment terms, or the list of acceptable means of payment (various credit cards, bank transfer, check, cash, etc.). More often than not, this is also data that cannot be changed by the application itself but that is statically defined or altered using specific administrative tools.

In the second group, you will find data that usually doesn't change more often than once a day. These are things like a list of articles, prices, etc. This group can also contain data that is changed at certain intervals, like once a day or once every hour. It is only important that you know exactly when and how often this data is changed. The last group of operative data contains all your real transaction data, the "meat" of your business. This is usually the most heavily changed data in your application, like customer information, inventory levels, and orders.

You will see that, in practice, most of the tables in your database will contain data of the first two groups, and only few tables actually contain data that is changed on a regular basis by business transactions. If you notice during the analysis process that some tables in your system actually contain data of different categories, you can change the database model or simply treat the two kinds of data differently in your application. This happens, for example, if you originally decided to store your article's inventory levels (which change all the time) together with the static data in the same table. You would have to separate this (either in the database or just in your object model) into the "real" product information on one hand and the transactional information (the inventory level) on the other hand.

Based on these three categories, you can now quite easily define matching caching strategies. The decision of whether or not to cache static data is made quite easily: you can usually cache everything that fits in terms of memory usage. Equally easy is the decision of whether or not to cache the operative data of the third group: you shouldn't store this data in memory, but instead always read/write the data to shared storage, for example, a database. Of course, there are exceptions for very specific applications, for example, if you can create fixed assignments between a number of customers and a number of hosts. In these applications, for example, you would store the complete customer data for all customers with names from A to G on one server, the data from H to N on a second, and the data from O to Z on a third one. But for most general applications, you should assume that these kinds of data should not be cached.

The most complex category in terms of caching strategies is the second group of seldom changed data, as there are two different types. On one hand, you'll find information that is usually very stable for a longer period of time and that will be changed over a longer period of time,

but that will become active at a previously known reference date (for example, "price list valid from July 4, 2004 00:00 am"). As you know the effective date up front, data of this kind will usually be easy to cache. You just have to make sure to remove any old copies at the correct point in time.

The second kind of data in this group is more complex. This is data that is usually not changed, but once it is changed it has to become effective immediately. This includes user permissions or similar user or customer lockout flags. Normally, this flag will never be set, but if someone decides to lock out a certain user, you have to make absolutely sure that it is done *right now*. If you cache this kind of data, you have to make sure to provide a sound cache-invalidation strategy as well. You can, for example, expose an additional remoting service on each node in your cluster that can be used to explicitly tell every machine on its own (and not just "the cluster") to remove some pieces of data from its in-memory cache.[6]

■ **Note** Please keep in mind that this refers not only to explicit caches, but generally to any information that is stored in memory, no matter whether it is a *static* field or a private member of a remoting server in *Singleton* mode.

Summary

In this chapter, you've seen some of the most important best practices for developing .NET Remoting applications. This list is, of course, not complete, but it reflects the most common points I have encountered on consulting projects, and in personal e-mail exchanges with several hundred developers who are using .NET Remoting in their daily projects.

Apart from a detailed look at several scenarios for .NET Remoting, you've learned that there are also numerous cases for which remoting is not the right solution. I have briefly introduced you to UDP and MSMQ, which can be used for scalable delivery of asynchronous notifications.

At the end of this chapter, you've seen how easy it is to configure a Windows Network Load Balancing cluster, which provides your applications with transparent failover and the possibility to scale out.

In the following chapter, I'll discuss the most common causes—and solutions—for problems and issues when using .NET Remoting.

---

6. Starting with .NET 2.0 and SQL Server 2005, you will in addition find features to let your server application be notified whether critical data changes. For example, you will be able *subscribe* to a query like `select userid, locked_out from users` in a way that SQL Server automatically informs you whether any of the corresponding records have been changed.

CHAPTER 10

Troubleshooting .NET Remoting

Whenever you develop applications spanning multiple processes, machines, networks, and maybe even versions of the .NET runtime, you might have to deal with a number of potential development or deployment issues. In this chapter, I present to you the most common problems, their causes, and how to resolve them. Some of these issues and workarounds will also be mentioned in chapters covering related topics throughout the book, but here I want to provide an additional concise section that you can refer to during your debugging and troubleshooting efforts.

Debugging Hints

Before going into the details of these common mistakes, I'd like to show you how to debug remoting applications. As you've seen in the previous chapters, any .NET application can be a server for .NET Remoting. The recommended way, however, is to host your remoting components in Internet Information Server (IIS).

The following is a reiteration of what I stated in Chapter 4, repeated here in case you've come directly to this chapter instead for debugging and troubleshooting guidance.

To debug a server application in IIS, you can select Debug ➤ Processes and manually attach the debugger to the ASP.NET worker process. This process is called w3p.exe on machines running Windows Server 2003 and aspnet_wp.exe on Windows XP. If you host your remoting components in a Windows service, you can use the same Debug ➤ Processes command to attach to your running service.

For IIS-hosted applications, however, there is also an easier but not very obvious solution to debugging your remoting server. In essence, you have to "trick" Visual Studio .NET into attaching itself automatically to the ASP.NET worker process. To do this, you add a new ASP.NET Web application project to your solution. This project will host your remoting configuration information and server-side assemblies.

In this project, you can add the <system.runtime.remoting> section to your web.config file as discussed previously. To add server-side implementation and interface assemblies, you just have to reference them using the Add Reference dialog box. These will be automatically deployed to the correct subdirectory.

You must not delete the default WebForm1.aspx as this will be used by Visual Studio to start the debugger. Now as soon as you hit F5, VS .NET will open an Internet Explorer window that displays this empty WebForm1.aspx. You can ignore this IE window, but you must not close it, because this would stop the debugger. Instead, you can go to your server-side implementation code in VS .NET and simply set breakpoints, view variables, and so on as if you were running in a normal remoting host.

Manual Breakpoints

Especially when troubleshooting applications running as Windows services or inside another host like IIS, you might want to set explicit breakpoints that should be triggered without first attaching a debugger. You can do this by including a call to `System.Diagnostics.Debugger.Break()` inside your server-side code.

This allows you to create conditional breakpoints that, for example, will only be triggered when some application-specific condition is met.

```
class CustomerManager: MarshalByRefObject, IRemoteCustomerManager
{
   public Customer GetCustomer(int id)
   {
    if (id == 42)
    {
      System.Diagnostics.Debugger.Break();
    }

    // ... implementation removed
   }
}
```

Please note that a call to `Debugger.Break()` will execute even if your application has been compiled in Release mode. To adapt this behavior to be consistent with your generic debugging requirements, it is recommended that you include this code in an `#if/#endif` if you intend to keep this code in your application.

```
class CustomerManager: MarshalByRefObject, IRemoteCustomerManager
{
   public Customer GetCustomer(int id)
   {
    #if DEBUG
      if (id == 42)
      {
        System.Diagnostics.Debugger.Break();
      }
    #endif

    // ... implementation removed
   }
}
```

If you create code like this, build the application in Debug mode, and deploy it to IIS or another remoting host, you don't have to explicitly attach a debugger. As soon as the breakpoint condition is met and `Debugger.Break()` is invoked, the Just-In-Time Debugging dialog box shown in Figure 10-1 will be displayed.

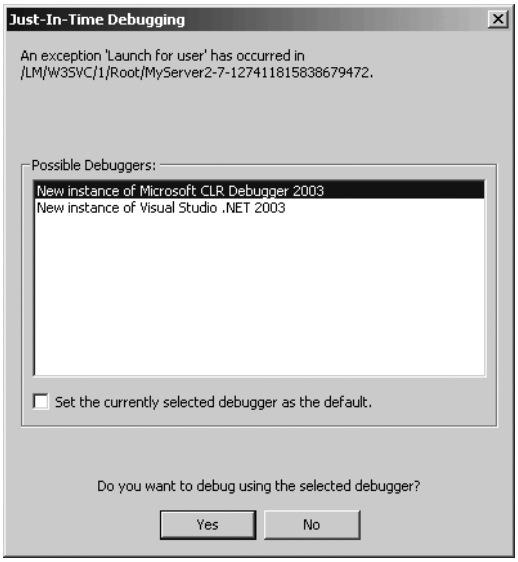
Just-In-Time Debugging
An exception 'Launch for user' has occurred in
/LM/W3SVC/1/Root/MyServer2-7-127411815838679472.
Possible Debuggers:
New instance of Microsoft CLR Debugger 2003
New instance of Visual Studio .NET 2003
Set the currently selected debugger as the default.
Do you want to debug using the selected debugger?
Yes
No

Figure 10-1. *Just-In-Time Debugging has been triggered.*

In this dialog box, you will see a list of debuggers that have been registered on your current system. You can now choose to debug using an instance of Visual Studio .NET or the CLR Debugger. The latter will start up considerably faster and can be especially interesting if you want to debug on a production system that contains the .NET SDK, but not Visual Studio .NET.

If you select No in this dialog box, the application will simply continue to run.

Configuration File Settings

A considerable number of .NET Remoting issues are caused by incorrect configuration settings. Because most of these settings are case sensitive, a minor typo will yield unexpected results. Configuration mistakes like these happen, for example, if you've chosen a deployment scenario in which you copy the complete server assembly to your client application and create your remoting proxies using a combination of `RemotingConfiguration.Configure()` and the `new` operator.[1]

Let's assume that you've created a class called CustomerManager in the namespace ServerImpl, and compiled it to a library called ServerImpl.DLL. In this case, the server-side configuration file may look similar to this one:

1. Please check the previous chapter on more reasons why this might not be a good idea in general.

```
<configuration>
  <system.runtime.remoting>
    <application>
      <channels>
        <channel ref="http" port="1234" />
      </channels>
      <service>
        <wellknown mode="Singleton"
               type="ServerImpl.CustomerManager, ServerImpl"
               objectUri="CustomerManager.rem" />
      </service>
    </application>
  </system.runtime.remoting>
</configuration>
```

If you were to now—mistakenly—create a client-side configuration like the following, your application would expose a strange behavior:

```
<configuration>
  <system.runtime.remoting>
    <application>
      <client>
        <wellknown type="ServerImpl.Customermanager, ServerImpl"
                   url="http://localhost:1234/CustomerManager.rem" />
      </client>
    </application>
  </system.runtime.remoting>
</configuration>
```

To complete this example, let's have a look at the respective snippet of your client application:

```
class Client
{
  static void Main(string[] args)
  {
     String filename =
         AppDomain.CurrentDomain.SetupInformation.ConfigurationFile;
     RemotingConfiguration.Configure(filename);

     CustomerManager mgr = new CustomerManager();
     Customer cust = mgr.GetCustomer(43);
  }
}
```

When the line `CustomerManager mgr = new CustomerManager()` is executed, you won't receive a remoting proxy pointing to your server as expected. Instead, the .NET Framework will create a local object of this type, and any further calls will be handled by the local object. The reason for this is that a small typo occurs in the client-side configuration file. Instead of specifying

"CustomerManager", you used "Customermanager". This is a mistake that happens quite often, especially because the .NET Remoting framework doesn't throw an exception during `RemotingConfiguration.Configure()` when it encounters an incorrectly specified *type* attribute. This usually leads to a larger debugging session, as your configuration file might contain a large number of configured types.

■ **Note** This is especially true because some remoting objects might even work if they are instantiated in your client application and not on the server as desired. They could, however, cease to work as soon as your application is deployed to your user's PC. A very common cause for this is a remoting component accessing a database, which might work directly on your (the developer's) machine because you have correct credentials to access your database server, but might not work on the user's machine because that user isn't known to SQL Server's integrated security system.

Local or Remote?

To verify whether or not a certain instance contains a "real" local object or a proxy pointing to a server, you can call `RemotingServices.IsTransparentProxy()` as illustrated in the following snippet:

```
String filename = AppDomain.CurrentDomain.SetupInformation.ConfigurationFile;
RemotingConfiguration.Configure(filename);
CustomerManager mgr = new CustomerManager();

if ( ! RemotingServices.IsTransparentProxy(mgr))
{
  throw new Exception("CustomerManager has not been correctly configured");
}
```

Checking each object after a call to `new` is, however, rather cumbersome and error prone, so you may want to use an alternative solution that checks the configured types after your application has read its remoting configuration file. To do this, you can simply iterate over the result of `RemotingConfiguration.GetRegisteredWellKnownClientTypes()`. For each WellKnownClientTypeEntry, you would then check whether its `ObjectType` property has been set to an existing type object. If this property is `null`, this means that the remoting framework could not find the type you have specified in your configuration file. In practice, this helps you to pinpoint potential typos in your client-side configuration file. This solution also works if you have chosen to use a purely interface-based method of remoting!

If you like to use this approach in your applications, you can simply follow a pattern similar to the following in your client-side code:

```
static void Main(string[] args)
{
  String filename = AppDomain.CurrentDomain.SetupInformation.ConfigurationFile;
  RemotingConfiguration.Configure(filename);
  VerifyRemotingConfiguration();
```

```
  CustomerManager mgr = new CustomerManager();
  // ... implementation removed ...
}

static void VerifyRemotingConfiguration()
{
  foreach (WellKnownClientTypeEntry en in
     RemotingConfiguration.GetRegisteredWellKnownClientTypes())
  {
    if (en.ObjectType == null)
    {
      throw new Exception("Could not find type " +
                en.TypeName + " in assembly " + en.AssemblyName);
    }
  }
}
```

The call to `VerifyRemotingConfiguration()` will only succeed if you've previously called `RemotingConfiguration.Configure()` and if all type attributes contained in this configuration file could be correctly resolved.

Checking Types on Your Server

A solution like the one presented previously, whereby you call a certain method to ensure a correct remoting configuration, unfortunately can't help you with your debugging efforts if you host your components in IIS. In this case, however, the solution is even easier: a special configuration setting, <debug>, can be used in server-side configuration files to verify the existence of all specified server-side type entries.

After including this setting in your server-side web.config file as shown in the following example, you will receive an exception as soon as the server tries to read this configuration file:

```
<configuration>
  <system.runtime.remoting>
    <application>
      <channels>
        <channel ref="http" port="1234" />
      </channels>
      <service>
        <wellknown mode="Singleton"
               type="ServerImpl.CustomerManager, ServerImpl"
               objectUri="CustomerManager.rem" />
      </service>
    </application>
    <debug loadTypes="true" />
  </system.runtime.remoting>
</configuration>
```

When contacting a server for which this setting has been included, but which contains a typo or misconfiguration in web.config, you'll receive the exception shown in Figure 10-2 on the client side. This is usually more desirable than hitting an exception later on when you try to access one of your many configured types.

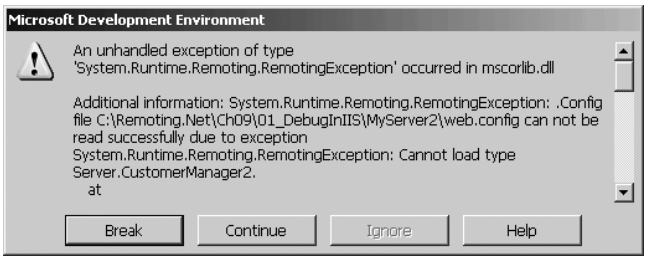

Figure 10-2. *Exception with <debug loadTypes="true" />*

BinaryFormatter Version Incompatibility

When hosting your server-side components in IIS while communicating with a BinaryFormatter (which is the recommended combination), you might experience a strange exception—a special form of the SerializationException as shown in Figure 10-3.

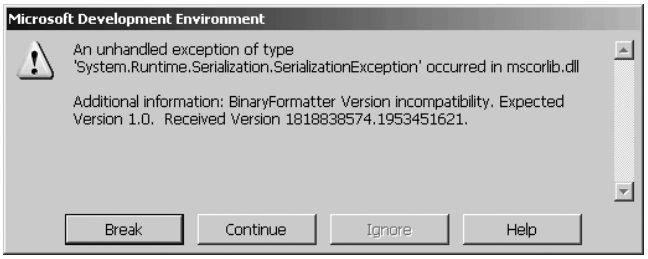

Figure 10-3. *The BinaryFormatter version incompatibility information*

I have to admit that I was quite stumped when this exception occurred for the first time in one of my applications. After all, I was sure that the version of the formatter—and indeed the version of the complete .NET Framework—was the same for the client and server machines.

When I started to investigate by using TcpTrace,[2] I looked at the response message from IIS, which you can see in Figure 10-4.

2. Free from `http://www.pocketsoap.com`.

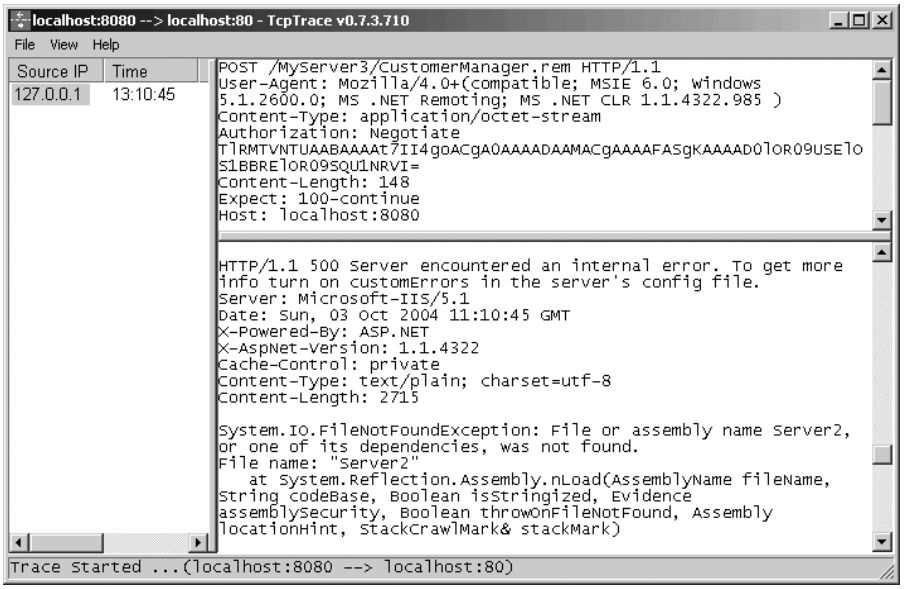

Figure 10-4. *The BinaryFormatter version incompatibility information in TcpTrace*

As you can see in the bottom-right window, the actual response message from server to client is a System.IO.FileNotFoundException. After more investigation about why the BinaryFormatter would interpret this as a version incompatibility, I noticed that the formatter simply does not look at the content type. When you host remoting components in IIS, you can basically get three different response types: text/plain, text/HTML, and application/octet stream. The first two will be returned if some misconfiguration triggers an internal server error in IIS. Only the latter, however, really contains a binary stream that can be read by the BinaryFormatter.

The formatter, however, does not check the content-type header, and always tries to deserialize the response stream according to its internal binary format definition. The formatter starts with reading the first few bytes from the stream to verify the version numbers. Somehow the string "System.IO.FileNotFoundException" is therefore interpreted as the version number 1818838574.1953451621. When the formatter compares this to the expected version number 1.0, it notices the mismatch and throws the corresponding exception.

Troubleshooting with a Custom Sink

As you will see in the second part of this book starting with the next chapter, the .NET Remoting framework provides an extensive set of extensibility hooks. This allows you to use a so-called *custom sink* to intercept the response message before it reaches the binary formatter. This sink—whose source code you will find in Chapter 11—is called *HttpErrorInterceptor* and will react according to the Content-Type header. To use it, you only have to modify your client-side configuration file to include an additional <provider> element.

```
<configuration>
   <system.runtime.remoting>
      <application>
         <channels>
```

```
            <channel ref="http" >
               <clientProviders>
                  <formatter ref="binary" />
                  <provider
    type="HttpErrorInterceptor.InterceptorSinkProvider, HttpErrorInterceptor" />
               </clientProviders>
            </channel>
         </channels>
   <!-- .... details removed .... -->
      </application>
   </system.runtime.remoting>
</configuration>
```

As soon as you use this configuration file, you will receive correct error messages as shown in Figure 10-5.

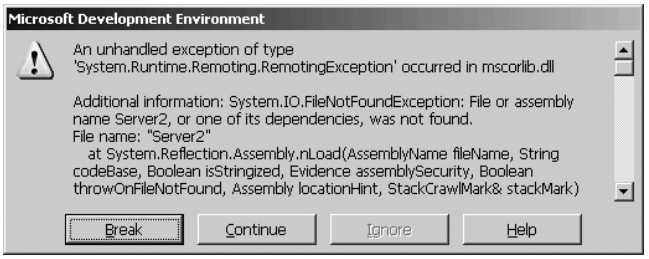

Figure 10-5. *The custom sink produces correct error messages.*

Changing Security Restrictions with TypeFilterLevel

If you pass objects that derive from MarshalByRefObject as method parameters to a remote server-side object without any further attention, you will trigger a security exception. The same thing is true whenever you subscribe to remote events on a server or register as a sponsor for a server-side object. You can see this exception in Figure 10-6.

Figure 10-6. *Security restrictions prevent sponsors and events.*

■ **Note** I include this here, in the *troubleshooting* chapter of this book, because this problem happens quite frequently when you take code from the Internet or from other sources that have been written for version 1.0 of the .NET Framework. The typeFilterLevel has been introduced only with version 1.1.

The reason for this exception is that these three operations can cause security or stability problems in remoting servers. In the first case, a malicious remote user could pass an incorrect URL to your remoting components that could point to resources inside a protected intranet zone. This way, the attacker might be able to invoke operations on internal servers that are not directly accessible to him or her.

For events and delegates, ,this same problem is true, but a malicious user might in addition negatively affect stability, performance, or scalability of your server by registering multiple—maybe even nonexistent—sponsors or events with your server-side objects. Whenever one of these implicit callbacks is invoked, the server will have to wait for all registered clients to acknowledge its requests. A malicious client could just keep the connection open without ever sending any response and therefore tie up valuable server-side resources. As remoting by default uses 25 threads per CPU, this would mean that a malicious client would only need to keep 25 callback connections open to halt your server.

This stability risk *only* applies to callbacks, events, and sponsors but not to regular method calls. A client only has the power to halt a server whenever the server invokes one of these callbacks. That's why you've received the exception mentioned earlier: to protect your server.

However, if you decide that you can ultimately trust all your client applications—for example, because you are running in an intranet and use corresponding authentication and authorization before allowing a client to subscribe to events—you can disable this security check. To do so, you have to specify the property `typeFilterLevel` as "Full" (default is "Low") for your server-side formatters. You can do this using a configuration file like the following:

```
<configuration>
  <system.runtime.remoting>
    <application>
      <channels>
        <channel ref="http" port="1234" >
               <serverProviders>
                  <formatter ref="binary" typeFilterLevel="Full" />
               </serverProviders>
               <clientProviders>
                  <formatter ref="binary" />
               </clientProviders>
            </channel>
      </channels>
      <service>
        <wellknown mode="Singleton"
               type="Server.RemoteFactory, Server"
               objectUri="Factory.rem" />
      </service>
    </application>
  </system.runtime.remoting>
</configuration>
```

■ **Note** Please note that this is a server-side configuration file that nevertheless includes a <clientProviders> section to specify which formatter should be used when invoking the callback to the client. As you can see, you do not need to specify `typeFilterLevel` for the client-side formatters.

Using Custom Exceptions

In the previous chapter, I discussed how custom exceptions have to be implemented so that you can use them in the remoting framework with cross-process method calls.

If you host these components in IIS, however, you have to be especially aware of the <customErrors> setting in your web.config file. This is a frequent source of confusion, because this configuration file may contain two different <customErrors> entries: one for general ASP.NET and one for remoting. However, they share a fundamental behavior: if the <customErrors> setting is set to "on", custom exception information and stack traces will not be sent to clients. If it is set to "remoteOnly", then clients from *localhost* will receive the complete exception information, whereas remote users will receive filtered information. If "off" is entered as a value for this setting, filtered exception information is sent to all clients. This also means that they will, for example, not receive any stack traces. The default value is "remoteOnly".

```
<configuration>
  <system.runtime.remoting>
    <customErrors mode="off" />
    <application>
      <service>
        <wellknown mode="Singleton"
                   type="Server.ExceptionTest, Server"
                   objectUri="ExceptionTest.rem" />
      </service>
    </application>
  </system.runtime.remoting>
</configuration>
```

Please note that this setting is configured directly underneath <system.runtime.remoting> and not underneath <application> like most other remoting-specific settings. Please also note that this setting for <customErrors> is not the same as the one in <system.web>. The latter only affects ASP.NET applications.

■ **Note** The naming of the setting "customErrors" can be slightly confusing. "Custom" in this regard means that the framework intercepts the exceptions thrown by an application and supplies "custom" information (which contains less data) instead. Here "on" means to filter all exceptions, whereas "off" allows the application-specific exceptions to pass through to the caller.

When a custom exception is thrown by your application, you will afterwards receive either the information shown in Figure 10-7, if customErrors has been set to "off", or the information shown in Figure 10-8, with customErrors set to "on."

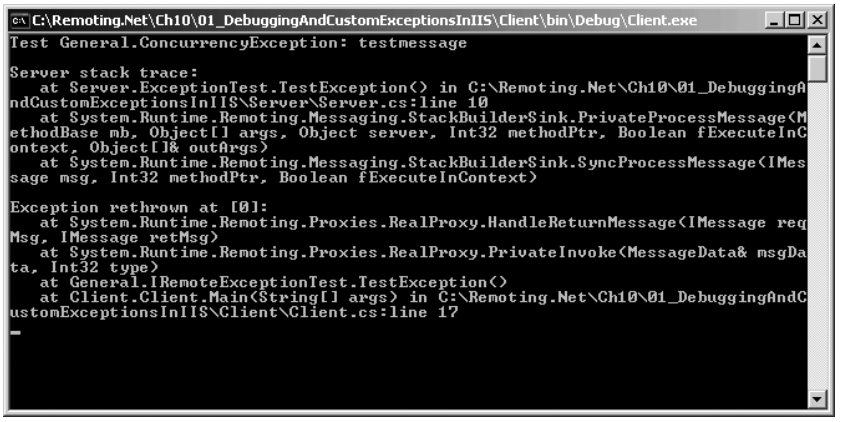
C:\Remoting.Net\Ch10\01_DebuggingAndCustomExceptionsInIIS\Client\bin\Debug\Client.exe
Test General.ConcurrencyException: testmessage
Server stack trace:
at Server.ExceptionTest.TestException() in C:\Remoting.Net\Ch10\01_DebuggingAndCustomExceptionsInIIS\Server\Server.cs:line 10
at System.Runtime.Remoting.Messaging.StackBuilderSink.PrivateProcessMessage(MethodBase mb, Object[] args, Object server, Int32 methodPtr, Boolean fExecuteInContext, Object[]& outArgs)
at System.Runtime.Remoting.Messaging.StackBuilderSink.SyncProcessMessage(IMessage msg, Int32 methodPtr, Boolean fExecuteInContext)
Exception rethrown at [0]:
at System.Runtime.Remoting.Proxies.RealProxy.HandleReturnMessage(IMessage reqMsg, IMessage retMsg)
at System.Runtime.Remoting.Proxies.RealProxy.PrivateInvoke(MessageData& msgData, Int32 type)
at General.IRemoteExceptionTest.TestException()
at Client.Client.Main(String[] args) in C:\Remoting.Net\Ch10\01_DebuggingAndCustomExceptionsInIIS\Client\Client.cs:line 17

Figure 10-7. *Detailed exception information with <customErrors mode="off" />*

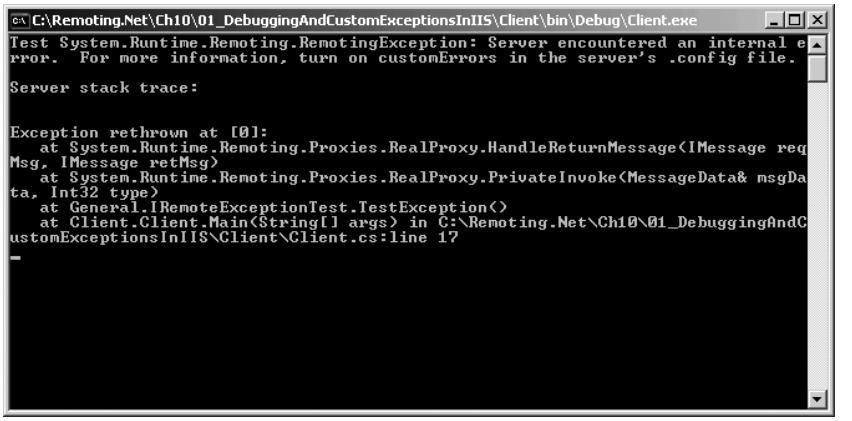
C:\Remoting.Net\Ch10\01_DebuggingAndCustomExceptionsInIIS\Client\bin\Debug\Client.exe
Test System.Runtime.Remoting.RemotingException: Server encountered an internal error. For more information, turn on customErrors in the server's .config file.
Server stack trace:
Exception rethrown at [0]:
at System.Runtime.Remoting.Proxies.RealProxy.HandleReturnMessage(IMessage reqMsg, IMessage retMsg)
at System.Runtime.Remoting.Proxies.RealProxy.PrivateInvoke(MessageData& msgData, Int32 type)
at General.IRemoteExceptionTest.TestException()
at Client.Client.Main(String[] args) in C:\Remoting.Net\Ch10\01_DebuggingAndCustomExceptionsInIIS\Client\Client.cs:line 17

Figure 10-8. *The same exception with <customErrors mode="on" />*

■ **Caution** In a try/catch block, the first exception will be of the "real" type thrown at the server-side application, whereas in the second case you will receive a generic RemotingException. You have to take this into account when designing exception-handling strategies.

Multihomed Machines and Firewalls

A multihomed machine is a computer that contains more than one network interface, or, in the context of remoting, more than one IP address. This includes routers, firewalls, and also servers that act as VPN gateways or that are connected to a dial-up line.

If your remoting services run on a multihomed machine, you might have to take some extra precautions to enable correct functionality. If your application only contains server-activated objects and if you never pass MarshalByRefObjects method parameters or return values, then you do *not* need the following.

To support client-activated objects and MarshalByRefObject as ref/out parameters for your return values, you have to specify the server's IP address or machine name in the configuration file. The reason for this is that, whenever you return a MarshalByRefObject, the server will return the complete URL for the given object to the client. The client will in turn use this URL to contact the remote object in subsequent calls.

To illustrate this, let's have a look at the data that is returned from the server when you create a client-activated object with a factory pattern. To do so, the example reuses the following interfaces to the remote components originally presented in Chapter 3:

```
public interface IRemoteFactory
{
  IRemoteObject GetNewInstance();
}

public interface IRemoteObject
{
  // ... removed
}
```

The implementation of the server-side IRemoteFactory just returns a new instance of a remote MarshalByRefObject.

```
class MyRemoteFactory: MarshalByRefObject,IRemoteFactory
{
  public IRemoteObject GetNewInstance()
  {
    return new MyRemoteObject();
  }
}

class MyRemoteObject: MarshalByRefObject, IRemoteObject
{
  // ... removed
}
```

When a client calls `GetNewInstance()` on the factory, the server will create a new object instance and *marshal* it to the client's process upon return on the method call. During this marshalling process, an ObjRef will be created that looks similar to the following (note though that this sample is simplified):

```
<ObjRef>
  <uri>/a34e3c81_66de_44a5_93c8_283add24d2cd/jhv+d3JpXxV69M431d+cInTK_1.rem</uri>
  <!-- parts removed -->
</ObjRef>
<!-- parts removed -->
<SOAP-ENC:Array id="ref-14" SOAP-ENC:arrayType="xsd:string[1]">
     <item>http://192.168.0.54:1234</item>
</SOAP-ENC:Array>
```

The complete URL to a remote object consists of the destination channel's base URL as shown in the `<item>` element at the end of this capture. The value that is contained in the ObjRef's `<uri>` property is subsequently appended to this base URL. As you can see, the server transfers its own IP address inside of this response packet.

Now imagine that your server has two IP addresses: an official Internet IP of 212.24.125.14 and an internal IP of 192.168.0.54. If you were to now marshal a MarshalByRefObject from this server to a client (for example, as a return value of a remote procedure call), the transferred ObjRef might either contain the internal or the external IP address. You may be tempted to expect that the remoting framework automatically uses the IP address on which the server has received the call to create this object, but this is not the case. Quite the contrary: you have to explicitly state which IP address or machine name should be returned to clients. Otherwise, a client outside your local network could receive the server's internal IP address and would not be able to call any method on the server-side objects.

To configure this setting, you can specify either the `bindTo` or the `machineName` property in your configuration file. In the first case, you can bind the remoting server to a specific IP address, whereas in the second case the system will use the specified machine name when returning MarshalByRefObjects to your clients. The framework does not check whether this machine name is associated with your machine or whether it even resolves to the same IP address as your server machine. This is a good thing as you will see shortly.

These two properties are specified on the <channel> level in your server-side configuration file, for example:

```
<channel ref="http" port="5555" bindTo="212.24.125.14" />
```

or

```
<channel ref="http" port="5555" machineName="yourserver.domain.com" />
```

Alternatively, you can use the following server-side code if you do not want to use configuration files:

```
Hashtable chnlProps = new Hashtable();
chnlProps["port"] = 1234;
chnlProps["machineName"] = "yourserver.domain.com";
// chnlProps["bindTo"] = "212.24.125.14";
HttpChannel chnl = new HttpChannel(chnlProps,null,null);
ChannelServices.RegisterChannel(chnl);
```

In the latter case, for example, the server will return the following information in the ObjRef (again, simplified):

```
<ObjRef>

  <uri>/a34e3c81_66de_44a5_93c8_283add24d2cd/jhv+d3JpXxV69M431d+cInTK_1.rem</uri>
  <!-- parts removed -->
</ObjRef>
<!-- parts removed -->
<SOAP-ENC:Array id="ref-14" SOAP-ENC:arrayType="xsd:string[1]">
     <item>http://yourserver.domain.com:1234</item>
</SOAP-ENC:Array>
```

When the client now tries to call a method of the returned MarshalByRefObject, it will contact it using the official address http://yourserver.domain.com:1234/.../..InTK_1.rem instead of trying to contact the nonroutable private IP address 192.168.0.54.

Client-Activated Objects Behind Firewalls

If you look at the preceding description, you will notice that it is also applicable to scenarios that involve firewalls, NATs with port forwarding, or reverse HTTP proxies. In all of these cases, a client will send its HTTP requests to an intermediary that will in turn forward it on to the actual Web server that is located in a DMZ as illustrated in Figure 10-9.

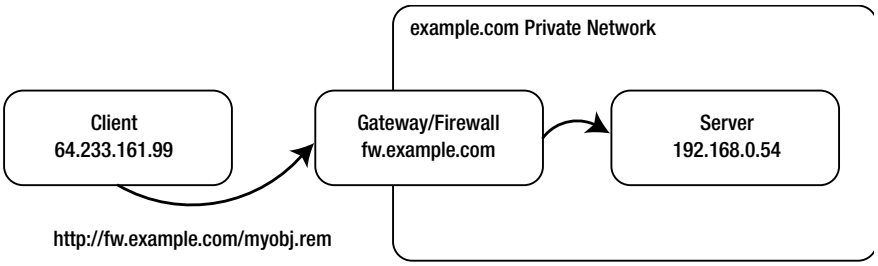

Figure 10-9. *Remoting with TCP port-forwarding firewalls*

In this case, the client directs its request to the gateway at fw.example.com and does not even know that this is not the final destination. The gateway can use simple TCP-based port forwarding to forward all incoming requests on port 80 (the default HTTP port) to the same port on 192.168.0.54.

In this case, you would have to specify the `machineName` property as described earlier if you use client-activated objects. Otherwise, the server would again return its internal IP address, which is unreachable from any machine outside its private network. In this case, you have to set this property to the name of the external gateway, i.e., to the only host the client can actually reach.

```
<channel ref="http" port="80" machineName="fw.example.com" />
```

■ **Note** You have to set the same port number on your firewall/gateway as you use on the destination machine, as it is not possible to dynamically adjust the port number that is returned as part of the ObjRef.

Summary

The .NET Remoting framework offers a big number of features with a relatively small number of "well-known pitfalls." In this chapter, I've illustrated the most common issues that turn up when remoting applications are developed or deployed.

You've seen how to debug different kinds of remoting application, and how to make sure that your configuration files are correct. This is especially important because most remoting-related problems stem from incorrect configuration files—a small typo or case mismatch in them can result in several hours of searching for a bug.

In addition, you've learned how to enable custom exceptions when hosting in IIS, and how to use client-activated objects behind firewalls.

This chapter is the last of the first part of this book. In the next five chapters, I'll show you how you can hook into the .NET Remoting framework to intercept remote procedure calls or even implement your own transport protocols.

PART 2

Extending

CHAPTER 11

Inside the Framework

As I stated in the introduction to this book, .NET provides an unprecedented extensibility for the remoting framework. The layered architecture of the .NET Remoting framework can be customized by either completely replacing the existing functionality of a given tier or chaining new implementation with the baseline .NET features.

Before working on the framework and its extensibility, I really encourage you to get a thorough understanding of the existing layers and their inner workings in this architecture. This chapter will give you that information. **But be forewarned: this chapter contains some heavy stuff.** It shows you how .NET Remoting really works. Some of the underlying concepts are quite abstract, but you don't necessarily need to know them if you just want to *use* .NET Remoting. If you want to *understand* or *extend* it, however, the information contained in this chapter is vital.

If you're only interested in the use of additional sinks, you'll find information pertaining to that topic in Chapter 12.

Looking at the Five Elements of Remoting

The .Net Remoting architecture, as you can see in Figure 11-1, is based on five core types of objects:

- *Proxies*: These objects masquerade as remote objects and forward calls.
- *Messages*: Message objects contain the necessary data to execute a remote method call.
- *Message sinks*: These objects allow custom processing of messages during a remote invocation.
- *Formatters*: These objects are message sinks as well and will serialize a message to a transfer format like SOAP.
- *Transport channels*: Message sinks yet again, these objects will transfer the serialized message to a remote process, for example, via HTTP.

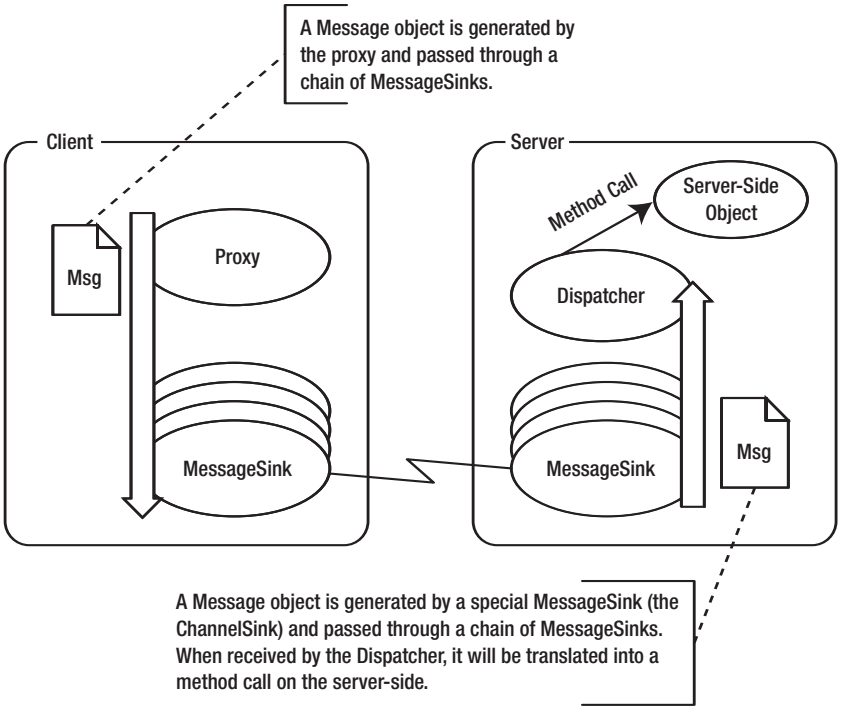
A Message object is generated by the proxy and passed through a chain of MessageSinks.
Client
Server
Method Call
Server-Side Object
Msg
Proxy
Dispatcher
MessageSink
MessageSink
Msg
A Message object is generated by a special MessageSink (the ChannelSink) and passed through a chain of MessageSinks. When received by the Dispatcher, it will be translated into a method call on the server-side.

Figure 11-1. *Simplified version of the .NET Remoting architecture*

A Bit About Proxies

Instead of dealing with "real" object references (memory references, pointers, and so on), when using remote objects, the client application can only perform methods on object *proxies*. These proxies masquerade, and therefore provide the same interface, as the target object. Instead of executing any method on their own, the proxies forward each method call to the .NET Remoting framework as a Message object.

This message then passes through the sinks shown previously, until it is finally handled by the server, where it passes through another set of message sinks until the call is placed on the "real" destination object. The server then creates a return message that will be passed back to the proxy. The proxy handles the return message and converts it to the eventual out/ref parameters and the method's return value, which will be passed back to the client application.

Creating Proxies

When using the new operator or calling `Activator.GetObject()` to acquire a reference to a remote object, the .NET Remoting framework generates two proxy objects. The first is an instance of the generic TransparentProxy (from System.Runtime.Remoting.Proxies). This is the object that will be returned from the new operator for a remote object.

Whenever you call a method on the reference to a remote object, you will in reality call it on this TransparentProxy. This proxy holds a reference to a RemotingProxy, which is a descendent of the abstract RealProxy class.

During the creation stage, references to the client-side message sink chains are acquired using the sink providers passed during channel creation (or the default, if no custom providers have been specified). These references are stored in the Identity object contained in the RealProxy.

After using the `new` operator or calling `GetObject()`, as shown in the following example, the variable `obj` will point to the TransparentProxy (see Figure 11-2):

```
HttpChannel channel = new HttpChannel();
ChannelServices.RegisterChannel(channel);
SomeClass obj = (SomeClass) Activator.GetObject(
      typeof(SomeClass),
      "http://localhost:1234/SomeSAO.soap");
```

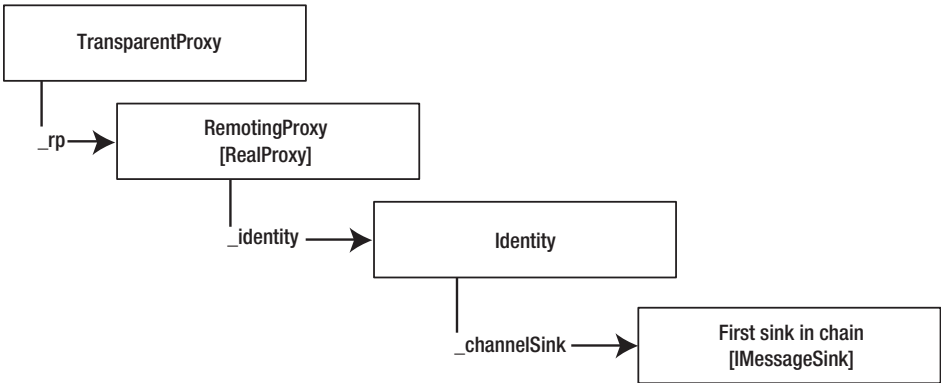

Figure 11-2. *Proxies with identity*

Creating Messages

When a call is placed on a remote object reference, the TransparentProxy creates a MessageData object and passes this to the RealProxy's `PrivateInvoke()` method. The RealProxy in turn generates a new Message object and calls `InitFields()`, passing the MessageData as a parameter. The Message object will now populate its properties by resolving the pointers inside the MessageData object.

For synchronous calls, the RealProxy places a chain of calls on itself, including `Invoke()`, `InternalInvoke()`, and `CallProcessMessage()`. The last one will look up the contained Identity object's sink chain and call `SyncProcessMessage()` on the first IMessageSink.

When the processing (including server-side handling) has completed, the call to this method will return an IMessage object containing the response message. The RealProxy will call its own `HandleReturnMessage()` method, which checks for out/ref parameters and will call `PropagateOutParameters()` on the Message object.

You can see a part of this client-side process when using the default HTTP channel, shown in Figure 11-3. If the client were to use the TCP channel instead, the channel would consist of a BinaryClientFormatterSink and a TcpClientTransport sink.

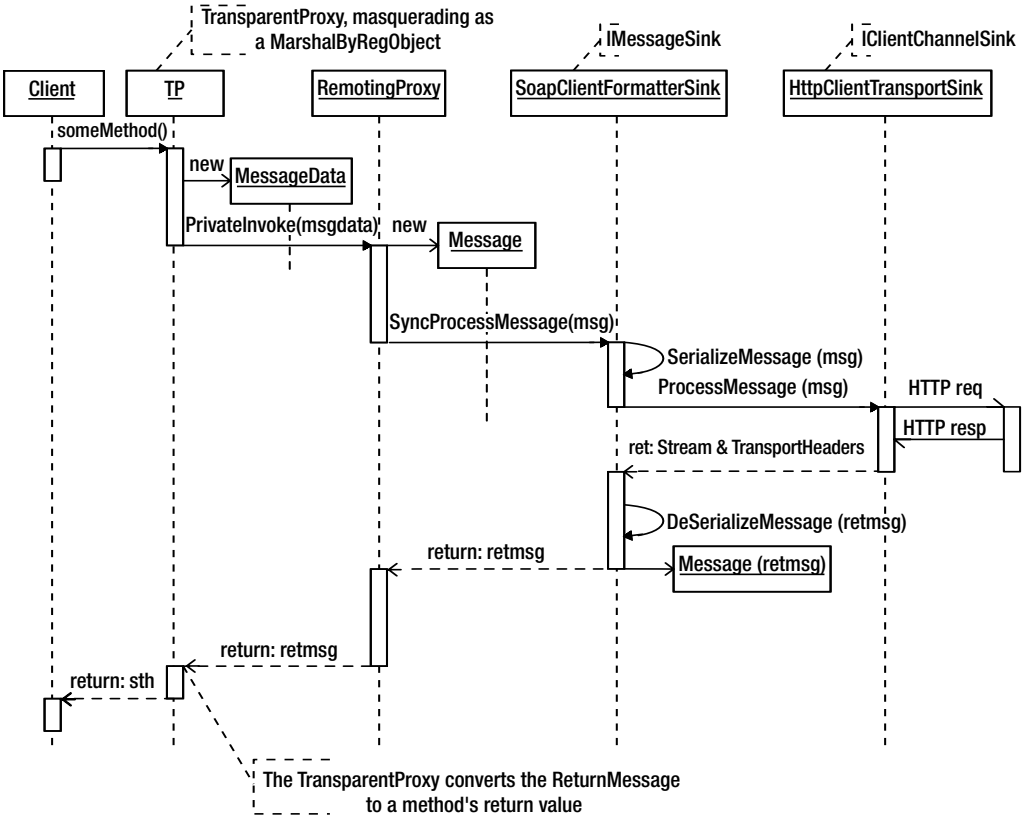

Figure 11-3. *Client-side synchronous message handling (partial)*

Returning Values

After this handling of the response, the RealProxy will return from its `PrivateInvoke()` method, and the IMessage is passed back to the TransparentProxy. Now a lot of "magic" happens behind the scenes: the TransparentProxy will take the return values and out parameters from the Message object and return them to the calling application in the conventional, stack-based method return fashion of the CLR. From the perspective of the client application, it will look as if a normal method call has just returned.

A Bit About the ObjRef Object

When working with CAOs, the client needs the capability to identify distinct object instances. When passing references to CAOs from one process to another, this identity information has to "travel" with the call. This information is stored in the serializable ObjRef object.

When instantiating a CAO (either by using a custom factory SAO as shown in Chapter 3 or by using the default RemoteActivation.rem SAO that is provided by the framework when the CAO is registered at the server side), a serialized ObjRef will be returned by the server.

This ObjRef is taken as a base to initialize the Identity object, and a reference to it will also be kept by the Identity. The ObjRef stores the unique URL to the CAO. This URL is based on a GUID and looks like the following:

```
/b8c0c989_68be_40d6_97b2_0c3fda5bb7ad/1014675796_1.rem
```

Additionally, the ObjRef object stores the server's base URL, which has also been returned by the server.

■ **Caution** This behavior is *very* different from that of SAOs, whereby the URL to the server is specified at the client. With CAOs, only the URL needed for the creation of the CAO is known to the client. The real connection endpoint URL for the CAO will be returned when creating the object. This can also mean that a host that's behind a firewall *might* return its private IP address and will therefore render the CAO unusable. To prevent this behavior, make sure to use either the machineName or bindTo attribute in the channel's configuration section as shown in Chapter 4.

You can check the ObjRef (or any other properties of the proxies) in a sample project simply by setting a breakpoint after the line where you acquire the remote reference of a CAO (for example, `SomeObj obj1 = new SomeObj()` when using configuration files). You can then open the Locals window and browse to the obj1's properties as show in Figure 11-4.

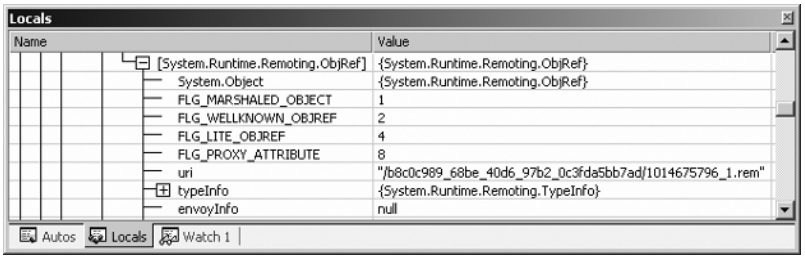

Figure 11-4. *Browsing to the ObjRef's properties in the Locals window*

You'll find the ObjRef shown in Figure 11-5 at obj1/_TransparentProxy/_rp/_identity/_objRef.

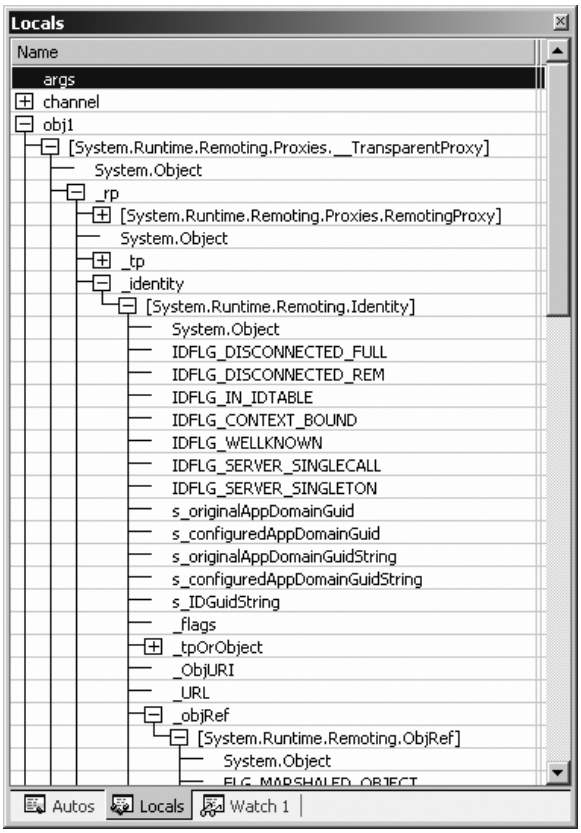

Figure 11-5. *Locating the ObjRef in the Locals window*

Understanding the Role of Messages

A *message* is basically just a dictionary object hidden behind the IMessage interface. Even though every message is based on this interface, the .NET Framework defines several special types thereof. You'll come across ConstructionCall and MethodCall messages (plus their respective return messages). The main difference between these message types is a predefinition of several entries in the internal dictionary.

While traveling through the chain of sinks, the message passes at least two important points: a formatter and a transport channel. The formatter is a special kind of sink that encodes the internal dictionary into some sort of wire protocol such as SOAP or a binary representation.

The transport channel will transfer a serialized message from one process to another. At the destination, the message's dictionary is restored from the wire protocol by a server-side formatter. After this, it passes through several server-side MessageSinks until it reaches the dispatcher. The dispatcher converts the message into a "real" stack-based method call that will be executed upon the target object. After execution, a return message is generated for most call types (excluding one-way calls) and passed back through the various sinks and channels until it reaches the client-side proxy, where it will be converted to the respective return value or exception.

What's in a Message?

There are several kinds of messages, and each of them is represented by a distinct class, depending on what kind of call it stands for. This object implements the IDictionary interface to provide key/value-based access to its properties.

A partial definition of MethodCall is shown here:

```
public class System.Runtime.Remoting.Messaging.MethodCall
{
    // only properties are shown

    public int ArgCount { virtual get; }
    public object[] Args { virtual get; }
    public bool HasVarArgs { virtual get; }
    public int InArgCount { virtual get; }
    public object[] InArgs { virtual get; }
    public LogicalCallContext LogicalCallContext { virtual get; }
    public MethodBase MethodBase { virtual get; }
    public string MethodName { virtual get; }
    public object MethodSignature { virtual get; }
    public IDictionary Properties { virtual get; }
    public string TypeName { virtual get; }
    public string Uri { virtual get; set; }
}
```

These values can be accessed in two ways. The first is by directly referencing the properties from the message object, as in `methodname = msg.MethodName`. The second way is to access the properties using the IDictionary interface with one of the predefined keys shown in the table that follows.

When doing this, a wrapper object (for example a MCMDictionary for MethodCallMessages) will be generated. This wrapper has a reference to the original message so that it can resolve a call to its dictionary values by providing the data from the underlying Message object's properties. Here you will see the dictionary keys and corresponding properties for a sample method call message:

Dictionary Key	Message's Property	Data Type	Sample Value
__Uri	Uri	String	/MyRemoteObject.soap
__MethodName	MethodName	String	setValue
__MethodSignature	MethodSignature	Object	*null*
__TypeName	TypeName	String	General.BaseRemote Object, General
__Args	Args	Object[]	{42}
__CallContext	LogicalCallContext	Object	*null*

The second kind of message, used during the instantiation of CAOs, is the ConstructionCall. This object extends MethodCall and provides the following additional properties:

```
public class System.Runtime.Remoting.Messaging.ConstructionCall
{
    // only properties are shown
    public Type ActivationType { virtual get; }
    public string ActivationTypeName { virtual get; }
    public IActivator Activator { virtual get; virtual set; }
    public object[] CallSiteActivationAttributes { virtual get; }
    public IList ContextProperties { virtual get; }
}
```

Examining Message Sinks

The transfer of a message from a client application to a server-side object is done by so-called *message sinks*. A sink will basically receive a message from another object, apply its own processing, and delegate any additional work to the next sink in a chain.

There are three basic interfaces for message sinks: IMessageSink, IClientChannelSink, and IServerChannelSink. As you can see in the following interface description, IMessageSink defines two methods for processing a message and a property getter for acquiring the reference for the next sink in the chain:

```
public interface IMessageSink
{

    IMessageSink NextSink { get; }

    IMessageCtrl AsyncProcessMessage(IMessage msg,
                                     IMessageSink replySink);

    IMessage SyncProcessMessage(IMessage msg);
}
```

Whenever an IMessageSink receives a message using either `SyncProcessMessage()` or `AsyncProcessMessage()`, it may first check whether it can handle this message. If it's able to do so, it will apply its own processing and afterwards pass the message on to the IMessageSink referenced in its `NextSink` property.

At some point in the chain, the message will reach a formatter (which is also an IMessageSink) that will serialize the message to a defined format and pass it on to a secondary chain of IClientChannelSink objects.

■ **Note** Formatters implement IClientFormatterSink by convention. This interface is a combination of IMessageSink and IClientChannelSink.

```
public interface IClientChannelSink
{

    // properties
    IClientChannelSink NextChannelSink { get; }
```

```
    // methods
    void AsyncProcessRequest(IClientChannelSinkStack sinkStack,
                             IMessage msg,
                             ITransportHeaders headers,
                             Stream stream);

    void AsyncProcessResponse(IClientResponseChannelSinkStack sinkStack,
                              object state,
                              ITransportHeaders headers,
                              Stream stream);

    Stream GetRequestStream(IMessage msg,
                            ITransportHeaders headers);

    void ProcessMessage(IMessage msg,
                        ITransportHeaders requestHeaders,
                        Stream requestStream,
                        ref ITransportHeaders responseHeaders,
                        ref Stream responseStream);
}
```

The main difference between IMessageSink and IClientChannelSink is that the former can access and change the original dictionary, independent of any serialization format, whereas the latter has access to the serialized message as a stream.

After processing the message, the IClientChannelSink also passes it on to the next sink in its chain until it reaches a transport channel like HttpClientTransportSink (which also implements IClientChannelSink) for the default HTTP channel.

Serialization Through Formatters

A Message object needs to be serialized into a stream before it can be transferred to a remote process, a task which is performed by a *formatter*. The .NET Remoting framework provides you with two default formatters, the SoapFormatter[1] and the BinaryFormatter, which can both be used via HTTP, IPC, or TCP connections.

■ **Note** In the samples that follow, you get a chance to take a look at the SoapFormatter, but the same information applies to BinaryFormatter (or any custom formatter) as well.

After the message completes the preprocessing stage by passing through the chain of IMessageSink objects, it will reach the formatter via the `SyncProcessMessage()` method.

---

1. Current prerelease information as of late 2004 mentions a possible future deprecation of the SoapFormatter with Version 2.0 of the .NET Framework.

On the client side, the SoapClientFormatterSink passes the IMessage on to its `SerializeMessage()` method. This function sets up the TransportHeaders and asks its `NextSink` (which will be the respective IClientChannelSink—that is, the HttpClientTransportSink) for the request stream onto which it should write the serialized data. If the request stream is not yet available, it will create a new ChunkedMemoryStream that will later be passed to the channel sink.

The real serialization is started from `CoreChannel.SerializeSoapMessage()`, which creates a SoapFormatter (from the System.Runtime.Serialization.Formatters.Soap namespace) and calls its `Serialize()` method.

You can see the SOAP output of the formatter for a sample call to `obj.setValue(42)` in the following excerpt. Remember that this is only the serialized form of the request—it is not yet transfer dependent (it does not contain any HTTP headers, for example).

```
<SOAP-ENV:Envelope
 xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
 xmlns:xsd="http://www.w3.org/2001/XMLSchema"
 xmlns:SOAP-ENC="http://schemas.xmlsoap.org/soap/encoding/"
 xmlns:SOAP-ENV="http://schemas.xmlsoap.org/soap/envelope/"
 SOAP-ENV:encodingStyle="http://schemas.xmlsoap.org/soap/encoding/"
 xmlns:i2="http://schemas.microsoft.com/clr/nsassem/General.BaseRemoteObj ➥
ect/General">
   <SOAP-ENV:Body>
     <i2:setValue id="ref-1">
        <newval>42</newval>
     </i2:setValue>
   </SOAP-ENV:Body>
</SOAP-ENV:Envelope>
```

Moving Messages Through Transport Channels

After the last IClientChannelSink (which can be either the formatter or custom channel sink) has been called, it forwards the message, stream, and headers to the `ProcessMessage()` method of the associated *transfer channel*. In addition to the stream generated by the formatter, this function needs an ITransportHeaders object, which has been populated by the formatter as well, as a parameter.

The transport sink's responsibility is to convert these headers into a protocol-dependent format—for example, into HTTP headers. It will then open a connection to the server (or check if it's is already open, for TCP channels or HTTP 1.1 KeepAlive connections) and send the headers and the stream's content over this connection.

Following the previous example, the HTTP headers for the SOAP remoting call will look like this:

```
POST /MyRemoteObject.soap HTTP/1.1
User-Agent: Mozilla/4.0+(compatible; MSIE 6.0; Windows 5.0.2195.0; MS .NET
Remoting;
MS .NET CLR 1.0.2914.16 )
SOAPAction:
"http://schemas.microsoft.com/clr/nsassem/General.BaseRemoteObject/General#
setValue
Content-Type: text/xml; charset="utf-8"
```

```
Content-Length: 510
Expect: 100-continue
Connection: Keep-Alive
Host: localhost
```

This leads to the following complete HTTP request for the `setValue(int)` method of a sample remote object:

```
POST /MyRemoteObject.soap HTTP/1.1
User-Agent: Mozilla/4.0+(compatible; MSIE 6.0; Windows 5.0.2195.0; MS .NET Remoting;
MS .NET CLR 1.0.2914.16 )
SOAPAction:
"http://schemas.microsoft.com/clr/nsassem/General.BaseRemoteObject/General#
setValue"
Content-Type: text/xml; charset="utf-8"
Content-Length: 510
Expect: 100-continue
Connection: Keep-Alive
Host: localhost

<SOAP-ENV:Envelope
   xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
   xmlns:xsd="http://www.w3.org/2001/XMLSchema"
   xmlns:SOAP-ENC="http://schemas.xmlsoap.org/soap/encoding/"
   xmlns:SOAP-ENV="http://schemas.xmlsoap.org/soap/envelope/"
   SOAP-ENV:encodingStyle="http://schemas.xmlsoap.org/soap/encoding/"
   xmlns:i2=
     "http://schemas.microsoft.com/clr/nsassem/General.BaseRemoteObject/General">
  <SOAP-ENV:Body>
    <i2:setValue id="ref-1">
      <newval>42</newval>
    </i2:setValue>
  </SOAP-ENV:Body>
</SOAP-ENV:Envelope>
```

I highlighted the important parts in this request so that you can see the values that have been taken from the message object's dictionary.

Client-Side Messaging

As you know by now, the message is created by the combination of the TransparentProxy and RemotingProxy, which send it to the first entry of the message sink chain. After this creation, the message will pass four to six stages, depending on the channel's configuration. You can see this in Figure 11-6.

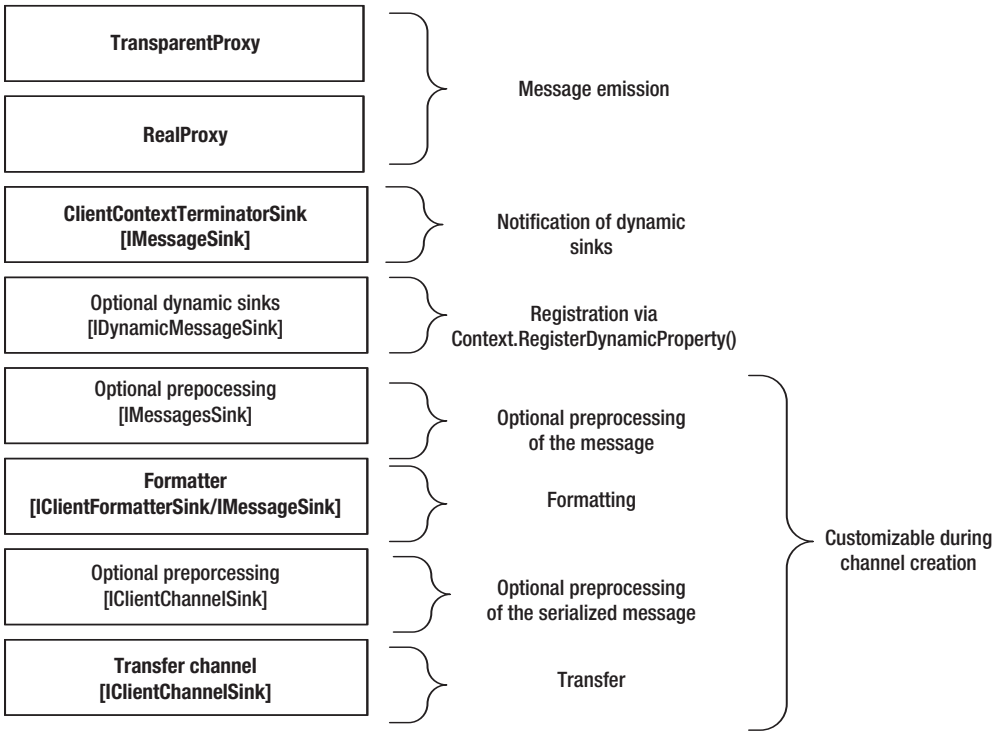

Figure 11-6. *Client-side layers and message stages*

The contents of the preprocessing, formatting, and transfer layers are customizable during the creation of the channel. When creating a default HTTP channel, for example, only a SoapClientFormatterSink (as the formatting layer) and an HttpClientTransportSink (as the transport layer) will be created; by default neither a preprocessing layer nor any dynamic context sinks are registered.

■ **Note** I show the processing of synchronous messages in these examples. Asynchronous messages will be handled later in this chapter.

ClientContextTerminatorSink and Dynamic Sinks

The ClientContextTerminatorSink is automatically registered for all channels. It is the first sink that gets called and in turn notifies any dynamic context sinks associated with the current remoting context. These dynamic sinks will receive the message via the IDynamicMessageSink interface.

```
public interface IDynamicMessageSink
{
    void ProcessMessageStart(IMessage reqMsg, bool bCliSide, bool bAsync);
    void ProcessMessageFinish(IMessage replyMsg, bool bCliSide, bool bAsync);
}
```

These sinks don't need to pass the information to any other sink in a chain. Instead, this is done automatically by the context terminator sink, which will call the relevant methods on a list of registered dynamic sinks before passing the message on to the next IMessageSink.

You'll find more on the creation and implementation of dynamic sinks in Chapter 12.

SoapClientFormatterSink

After passing through optional custom IMessageSink objects, the message reaches the formatter. As shown previously, the formatter's task is to take the message's internal dictionary and serialize it to a defined wire format. The output of the serialization is an object implementing ITransferHeaders and a stream from which the channel sink will be able to read the serialized data.

After generating these objects, the formatter calls `ProcessMessage()` on its assigned IClientChannelSink and as a result starts to pass the message to the secondary chain—the channel sink chain.

HttpClientChannel

At the end of the chain of client channel sinks, the message ultimately reaches the transfer channel, which also implements the IClientChannelSink interface. When the `ProcessMessage()` method of this channel sink is called, it opens a connection to the server (or uses an existing connection) and passes the data using the defined transfer protocol. The server now processes the request message and returns a ReturnMessage in serialized form. The client-side channel sink will take this data and split it into an ITransferHeader object, which contains the headers, and into a stream containing the serialized payload. These two objects are then returned as out parameters of the `ProcessMessage()` method.

After this splitting, the response message travels back through the chain of IClientChannelSinks until reaching the formatter, where it is deserialized and an IMessage object created. This object is then returned back through the chain of IMessageSinks until it reaches the two proxies. The TransparentProxy decodes the message, generates the method return value, and fills the respective out or ref parameters.

The original method call—which has been placed by the client application—then returns and the client now has access to the method's responses.

Server-Side Messaging

During the creation of the server-side channel (I use the HttpServerChannel in the following description, but the same applies to other server channels as well), the server-side sink chain is created and a TcpListener is spawned in a new thread. This object starts to listen on the specified port and notifies the HttpServerSocketHandler when a connection is made.

The message then passes through the layers shown in Figure 11-7.

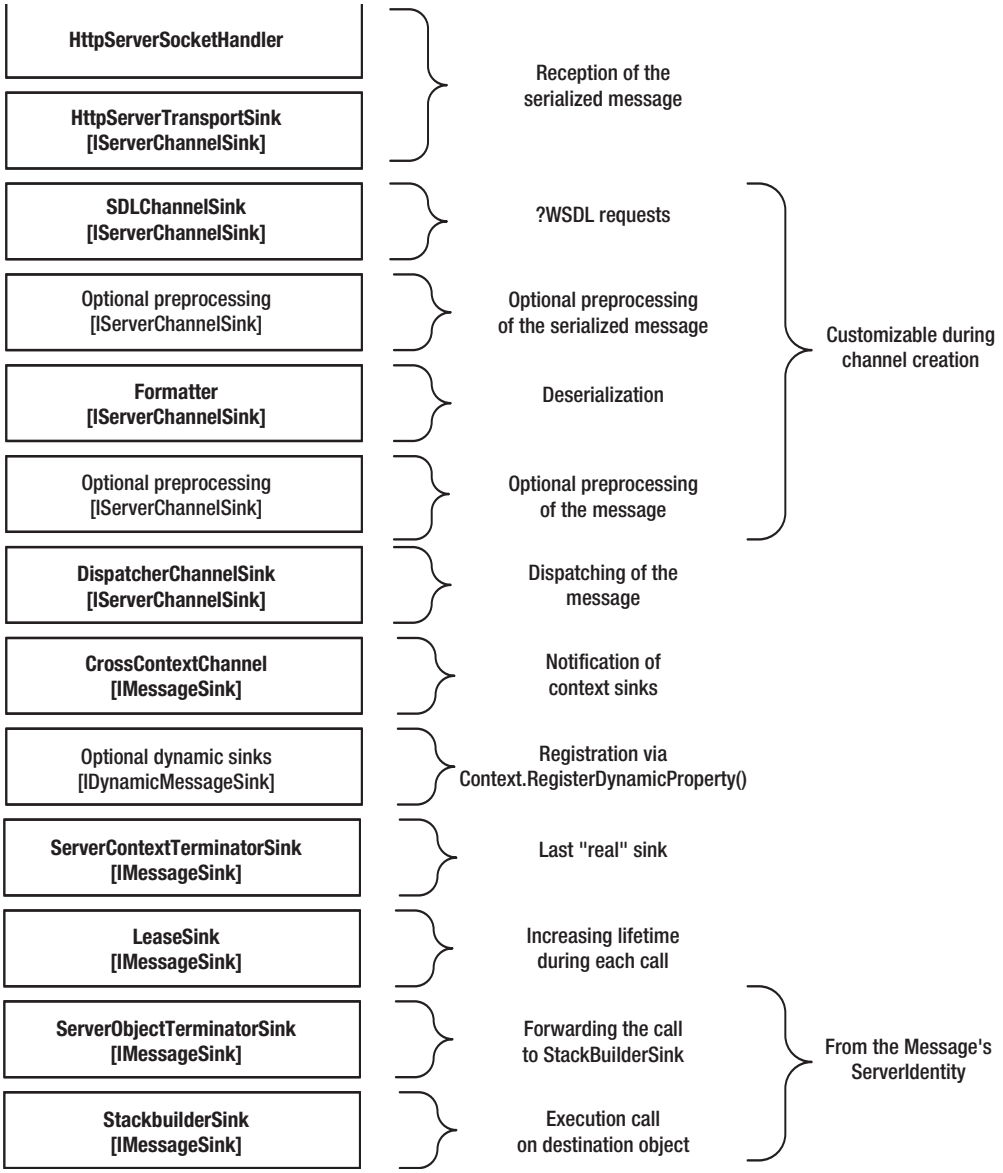

Figure 11-7. *Server-side messaging layers*

One of the main differences between client-side and server-side message processing is that on the server side, each sink's `ProcessMessage()` method takes a parameter of type ServerChannelSinkStack. Every sink that is participating in a call pushes itself onto this stack before forwarding the call to the next sink in the chain. The reason for this is that the sinks do not know up front if the call will be handled synchronously or asynchronously. Every sink that's been pushed onto the stack will get the chance to handle the asynchronous reply later.

You can see the IServerChannelSink interface here:

```
public interface IServerChannelSink
{

   IServerChannelSink NextChannelSink { get; }

   ServerProcessing ProcessMessage(IServerChannelSinkStack sinkStack,
                                   IMessage requestMsg,
                                   ITransportHeaders requestHeaders,
                                   Stream requestStream,
                                   ref IMessage responseMsg,
                                   ref ITransportHeaders responseHeaders,
                                   ref Stream responseStream);

   void AsyncProcessResponse(IServerResponseChannelSinkStack sinkStack,
                             object state,
                             IMessage msg,
                             ITransportHeaders headers,
                             Stream stream);

   Stream GetResponseStream(IServerResponseChannelSinkStack sinkStack,
                            object state,
                            IMessage msg,
                            ITransportHeaders headers);

}
```

HttpServerChannel and HttpServerTransportSink

When a connection to the server-side channel is opened, an instance of HttpServerSocketHandler is created and supplied with a delegate that points to the HttpServerTransportSink's `ServiceRequest()` method. This method will be called after the background thread finishes reading the request stream.

The HttpServerTransportSink sets up the ServerChannelSinkStack and pushes itself onto this stack before forwarding the call to the next sink.

After the chain has finished processing the message (that is, after the method call has been executed), it generates the HTTP response headers. These will be either "200 OK" for synchronous calls or "202 Accepted" for one-way messages.

SDLChannelSink

The SDLChannelSink is a very special kind of sink that really shows the power of the .NET Remoting framework's extensibility. Contrary to most other sinks, it does not forward any requests to the destination object, but instead generates the WSDL information needed for the creation of proxies.

It does this whenever it encounters either of the strings "?WSDL" or "?SDL" at the end of an HTTP GET request. In this case, the WSDL will be generated by calling the `ConvertTypesToSchemaToStream()` method from System.Runtime.Remoting.MetadataServices.MetaData.

■ **Note** MetaData is the same class SoapSuds uses when generating proxies.

When the HTTP request is of type POST *or* when it is a GET request that does not end with the string "?WSDL" or "?SDL", the message will be passed to the next sink.

SoapServerFormatterSink and BinaryServerFormatterSink

The default server-side HTTP channel uses both formatters in a chained fashion. The first formatter that's used is the SoapServerFormatterSink. This sink first checks whether the serialized message contained in the request stream is a SOAP message. If this is the case, it is deserialized, and the resulting IMessage object is passed to the next sink (which is BinaryServerFormatterSink). In this case, the stream (which is needed as an input parameter to the next sink) will be passed as null.

If the stream does not contain a SOAP-encoded message, it will be copied to a MemoryStream and passed to the BinaryServerFormatterSink.

The binary formatter employs the same logic and passes either the deserialized message (which it might have already gotten from the SOAP formatter, *or* which it can deserialize on its own) or the stream (if it cannot decode the message) to the next sink.

Both BinaryServerFormatterSink and SoapServerFormatterSink only push themselves onto the SinkStack (before calling `ProcessMessage()` on the subsequent sink) when they can handle the message. If neither BinaryServerFormatterSink nor SoapServerFormatterSink could deserialize the message *and* the next sink is not another formatter, an exception is thrown.

DispatchChannelSink

After passing through an optional layer of IMessageSink objects, the message reaches the dispatcher. The DispatchChannelSink takes the decoded IMessage and forwards it to `ChannelServices.DispatchMessage()`. This method checks for disconnected or timed-out objects and dynamically instantiates SAOs (Singleton or SingleCall) if they do not exist at the server.

After the possible creation of the necessary destination object, it promotes this call to `CrossContextChannel.SyncProcessMessage()` or `CrossContextChannel.AsyncProcessMessage()` if the call's target is a one-way method.

CrossContextChannel

The CrossContextChannel notifies dynamic context sinks and passes the IMessage on to ServerContextTerminatorSink. A dynamic sink does not implement IMessageSink, but rather the interface IDynamicMessageSink, which is shown here:

```
public interface IDynamicMessageSink
{
    void ProcessMessageStart(IMessage reqMsg, bool bCliSide, bool bAsync);
    void ProcessMessageFinish(IMessage replyMsg, bool bCliSide, bool bAsync);
}
```

The same logic is applied as with client-side dynamic sinks: these sinks do not have to call any additional sinks in the chain, as this will be taken care of by the framework.

The `ProcessMessageStart()` method is called *before* passing the message on to ServerContextTerminatorSink and `ProcessMesssageFinish()` is called after the call to the context terminator sink returns. Both methods may change the IMessage object's properties.

You can read more about dynamic sinks later in Chapter 12.

ServerContextTerminatorSink

This sink's behavior is probably the most complex one. It is the last "hardwired" sink, and therefore has no *direct* references to other sinks in the chain. So how can the method call specified in the message be executed?

The ServerContextTerminatorSink looks at the IMessage's ServerIdentity object, using `InternalSink.GetServerIdentity()`, and requests this object's message sink chain using `ServerIdentity.GetServerObjectChain()`.

The ServerIdentity's object chain is populated by a call to the static method `CreateServerObjectChain()` of the Context class. This call creates a ServerObjectTerminatorSink at the absolute end of the chain and puts other sinks before it. These other sinks are obtained from the DomainSpecificRemotingData object, which is held in the `RemotingData` property of AppDomain.

DomainSpecificRemotingData by default contains a reference to LeaseSink that will be placed in the chain *before* the terminator sink. The resulting sink chain, which is obtained from the ServerIdentity object, is shown in Figure 11-8.

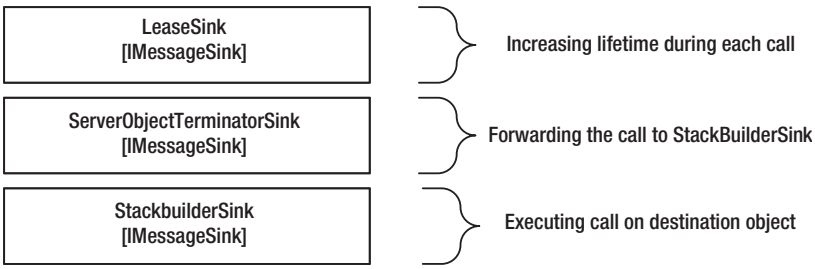

Figure 11-8. *Sinks from the ServerIdentity object*

LeaseSink

The lease sink will have a reference to the destination MarshalByRefObject's lease. It simply calls the `RenewOnCall()` method of this lease and passes the call on to the next sink. The `RenewOnCall()` method looks at the RenewOnCallTime configuration setting (default of two minutes) and sets the object's time to live to this value.

ServerObjectTerminatorSink and StackbuilderSink

The object terminator sink will forward the call to the StackBuilderSink. This final message sink is a kind of "magic" sink. First it checks if it's okay to remotely call the method specified in the IMessage object. It does this by checking whether the destination object matches the request, meaning that there is a "direct match," or the object implements the requested interfaces or is derived from the requested base class.

After verifying the message, the sink uses several external functions to create a stack frame (that is, it makes the transition between message-based execution and stack-based execution) and calls the destination method. It then generates a return message that contains the result and, if available, the values from any ref or out parameters. This ReturnMessage object is then returned from the call to `SyncProcessMessage()`.

All About Asynchronous Messaging

The previous part of this chapter only covers synchronous processing due to one reason: it's a lot easier and consistent between client and server than other types of processing.

As you know from the examples and figures earlier in this chapter, several methods are available for message handling: on the client side, there are the IMessageSink and the IClientChannelSink interfaces. Both sink types approach the handling of asynchronous messages in a substantially different manner, as described in the next sections.

Asynchronous IMessageSink Processing

When handling the messages in a synchronous manner in an IMessageSink chain, the response message will simply be the return value of the method call. You can see this in the following snippet, which shows a sample IMessageSink (I've omitted parts of the interface and only display the `SyncProcessMessage()` method here):

```
class MySink1: IMessageSink {
   IMessageSink _nextSink;
   IMessage SyncProcessMessage(IMessage msg) {

      // here you can do something with the msg

      IMessage retMsg = _nextSink.SyncProcessMessage(msg);

      // here you can do something with the retMsg

      // and then, simply return the retMsg to the previous sink
      return retMsg;
   }
}
```

When implementing asynchronous processing that is triggered whenever you use a Delegate's `BeginInvoke()` method, the call to `NextSink.AsyncProcessMessage()` is returned immediately. The response message is sent to a secondary chain, which is passed in the `replySink` parameter of the call to `AsyncProcessMessage()`.

First, I show you how to do asynchronous processing when you *don't* want to be notified of the call's return:

```
public class MySink1: IMessageSink {
   IMessageSink _nextSink;
   IMessageCtrl AsyncProcessMessage(IMessage msg,
                                    IMessageSink replySink)
   {
      // here you can do something with the msg
```

```
      return _nextSink.AsyncProcessMessage(msg, replySink);
   }
}
```

In `replySink`, you'll receive the first entry to a chain of IMessageSink objects that want to be notified upon completion of the asynchronous call.

If you want to handle the reply message in a sink of your own, you have to instantiate a new IMessageSink object and "chain" it to the existing list of reply sinks. You can see this in the following snippet (again, parts of the interface have been omitted):

```
public class MyReplySink: IMessageSink {
   IMessageSink _nextSink;

   MyReplySink(IMessageSink next) {
      // .ctor used to connect this sink to the chain
      _nextSink = next;
   }

   IMessage SyncProcessMessage(IMessage msg) {
      // the msg will be the reply message!

      // here you can do something with the msg

      // and then, pass it onto the next reply sink in the chain
      IMessage retMsg = _nextSink.SyncProcessMessage(msg);

      return retMsg;
   }
}

public class MySink1: IMessageSink {
   IMessageSink _nextSink;
   IMessageCtrl AsyncProcessMessage(IMessage msg,
                                    IMessageSink replySink)
   {
      // here you can do something with the msg

      // create a new reply sink which is chained to the existing replySink
      IMessageSink myReplyChain = new MyReplySink(replySink);

      // call the next sink's async processing
      return _nextSink.AsyncProcessMessage(msg, myReplyChain);
   }
}
```

When the async call is completed in this example, you'll have the option to change the reply message in the `MyReplySink.SyncProcessMessage()` method.

■ **Note** The reply message is processed synchronously; only the generation of the message happens asynchronously at the server.

Asynchronous IClientChannelSink Processing

As you can see in the following partial definition of the IClientChannelSink interface, there are, in contrast to the IMessageSink interface, distinct methods for handling the asynchronous request and reply:

```
public interface IClientChannelSink
{
    void AsyncProcessRequest(IClientChannelSinkStack sinkStack,
                             IMessage msg,
                             ITransportHeaders headers,
                             Stream stream);

    void AsyncProcessResponse(IClientResponseChannelSinkStack sinkStack,
                              object state,
                              ITransportHeaders headers,
                              Stream stream);
}
```

When the `AsyncProcessRequest()` method is called while an IMessage travels through a sink chain, it will receive an IClientChannelSinkStack as a parameter. This sink stack contains all sinks that want to be notified when the asynchronous processing returns. If your sink wants to be included in this notification, it has to push itself onto this stack before calling the next sink's `AsyncProcessRequest()` method.

You can see this in the following snippet (only parts are shown):

```
public class SomeSink: IClientChannelSink
{

      IClientChannelSink _nextChnlSink;

      public void AsyncProcessRequest(IClientChannelSinkStack sinkStack,
                                      IMessage msg,
                                      ITransportHeaders headers,
                                      Stream stream)
       {

         // here you can work with the message
```

```
        // pushing this sink onto the stack
        sinkStack.Push (this,null);

        // calling the next sink
        _nextChnlSink.AsyncProcessRequest(sinkStack,msg,headers,stream);
    }
}
```

The sinkStack object that is passed to `AsyncProcessRequest()` is of type ClientChannelSinkStack and implements the following IClientChannelSinkStack interface, which allows a sink to push itself onto the stack:

```
public interface IClientChannelSinkStack : IClientResponseChannelSinkStack
{
    object Pop(IClientChannelSink sink);
    void Push(IClientChannelSink sink, object state);
}
```

Whatever is being passed as the state parameter of `Push()` will be received by sink's `AsyncProcessResponse()` method upon completion of the call. It can contain any information the sink might need while processing the response. For built-in sinks, this will be the original message that triggered the request.

As the previous interface extends the IClientReponseChannelSinkStack interface, I'll show you this one here as well:

```
public interface IClientResponseChannelSinkStack
{
    void AsyncProcessResponse(ITransportHeaders headers, Stream stream);
    void DispatchException(Exception e);
    void DispatchReplyMessage(IMessage msg);
}
```

The `AsyncProcessResponse()` method of the ClientChannelSinkStack pops one sink from stack and calls this sink's `AsyncProcessResponse()` method. Therefore, the "reverse" chaining that uses the sink stack works simply by recursively calling the stack's `AsyncProcessResponse()` method from each sink. This is shown in the following piece of code:

```
public class SomeSink: IClientChannelSink
{
    public void AsyncProcessResponse(
                IClientResponseChannelSinkStack sinkStack,
                object state,
                ITransportHeaders headers,
                Stream stream)
     {

        // here you can work with the stream or headers
```

```
            // calling the next sink via the sink stack
            sinkStack.AsyncProcessResponse(headers,stream);
        }
}
```

Generating the Request

In the following text, I show you how an asynchronous request is generated and how the response is handled. The example I'll use is based on the following custom sinks:

- *MyMessageSink*: A custom IMessageSink
- *MyResponseSink*: A sink created by MyMessageSink as a handler for the asynchronous response
- *MyChannelSink*: A custom IClientChannelSink implementation

Regardless of whether the request is synchronous or asynchronous, the IMessageSink chain is handled *before* any calls to a IClientChannelSink. Immediately before the first call to `IClientChannelSink.AsyncProcessRequest()`, a ClientSinkStack object is instantiated and gets a reference to the IMessageSink reply chain. You can see the beginning of an asynchronous call in Figure 11-9.

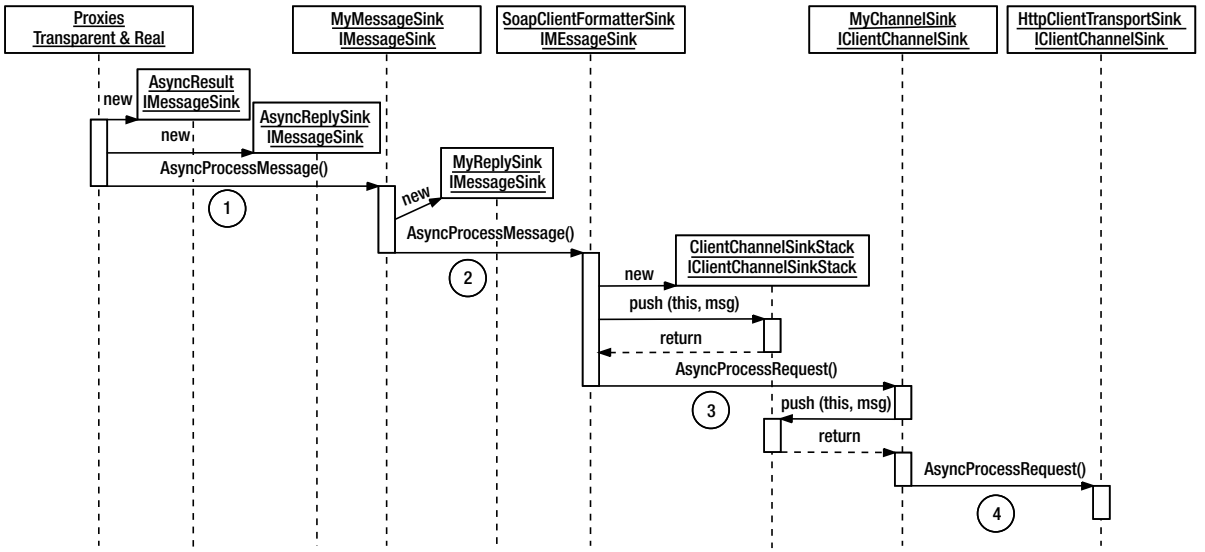

Figure 11-9. *First phase of an asynchronous call*

In Figures 11-10 through 11-13, you'll see the contents of the replySink and the SinkStack that are passed as parameters at the points marked (1) through (4) in Figure 11-9. In Figure 11-10 you can see the predefined reply chain that is established for each asynchronous call *before* reaching the first IMessageSink. The purpose of these sinks is to handle the asynchronous response in a form that's compatible with the "conventional" delegate mechanism.

The final sink for the asynchronous reply will be the AsyncResult object that is returned from the delegate's `BeginInvoke()` method.

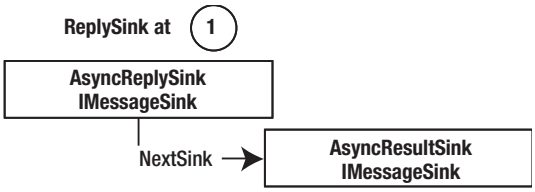

Figure 11-10. *SinkStack before call to first custom IMessageSink*

When MyMessageSink's `AsyncProcessResponse()` method is called, it generates a new reply sink, named MyReplySink, that is linked to the existing reply chain. You can see the ReplySink parameter that is passed to the next sink in Figure 11-11.

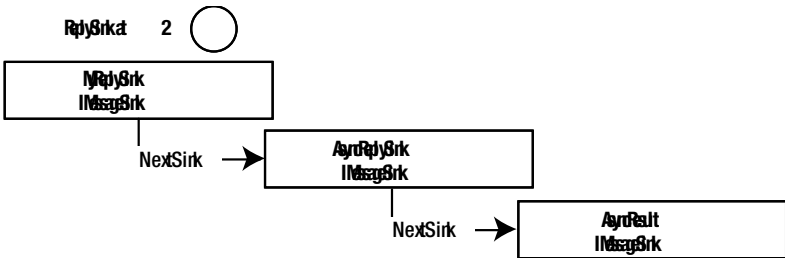

Figure 11-11. *SinkStack before call to SoapClientFormatterSink*

Figure 11-12 shows you the contents of the ClientChannelSinkStack after SoapClientFormatterSink has finished its processing. The stack contains a reference to the previous IMessageSink stack, shown in Figure 11-11, and points to the first entry in the stack of IClientFormatterSinks. This is quite interesting insofar as the SOAP formatter has been called as an IMessageSink but pushes itself onto the stack of IClientChannelSinks.

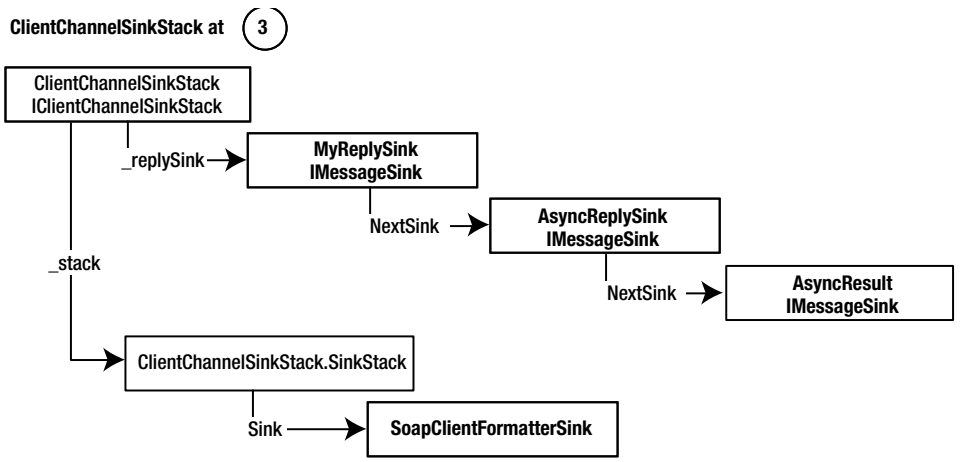

Figure 11-12. *SinkStack before call to first custom IClientChannelSink*

When the secondary custom IClientChannelSink object, MyChannelSink, is called, it pushes itself onto the stack and calls the `AsyncProcessRequest()` method of HttpClientTransportSink. In Figure 11-13 you can see the resulting channel sink stack before the HTTP request is sent.

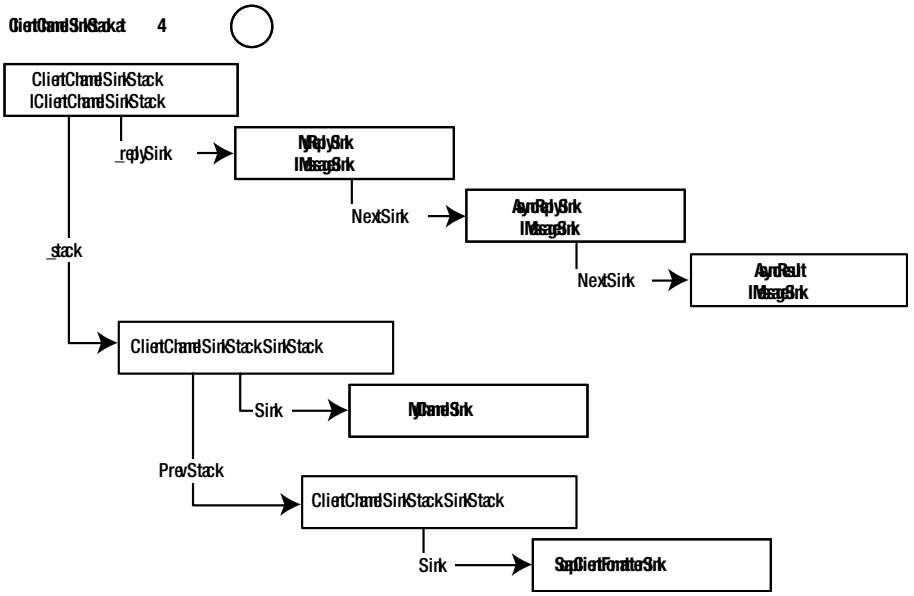

Figure 11-13. *SinkStack before call to HttpClientTransportSink*

Handling the Response

On receiving the HTTP response, HttpClientTransportSink calls `AsyncProcessResponse()` on the ClientChannelSinkStack. The sink stack then pops the first entry from the stack (using a different implementation, as with its `public Pop()` method) and calls `AsyncProcessResponse()` on the IClientChannelSink that is at the top of the stack. You can see the sequence of calls that follow in Figure 11-14.

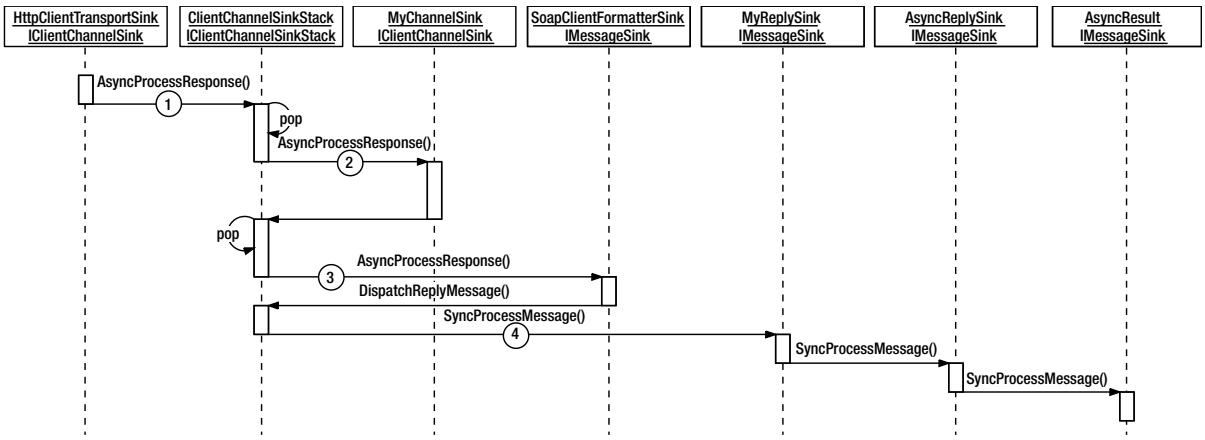

Figure 11-14. *Handling an asynchronous response*

Before this call—the point marked with (1) in the diagram—the ClientChannelSinkStack will look the same as in Figure 11-13. You can see the state of this stack after the "pop" operation in Figure 11-15.

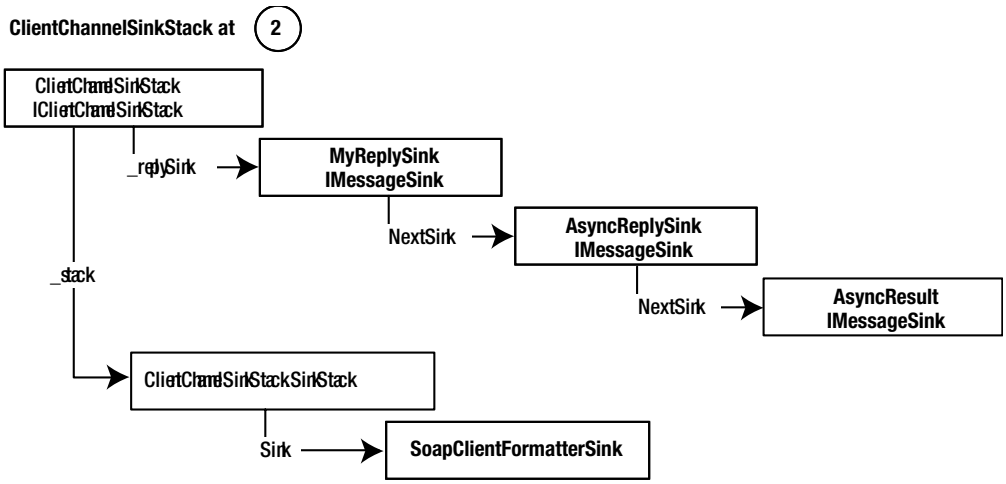

Figure 11-15. *The ClientChannelSinkStack before the first sink is called*

In the following step, the sink stack places a call to the custom MyChannelSink object. This sink will handle the call as shown in the following source code fragment and will therefore just proceed with invoking `AsyncProcessResponse()` on the sink stack again:

```
public void AsyncProcessResponse(IClientResponseChannelSinkStack sinkStack,
                                 object state,
                                 ITransportHeaders headers,
                                 Stream stream)
{
      sinkStack.AsyncProcessResponse(headers,stream);
}
```

The ClientChannelSinkStack now pops the next sink from the internal stack and forwards the call on to this SoapClientFormatterSink. You can see the state of the stack at Figure 11-16.

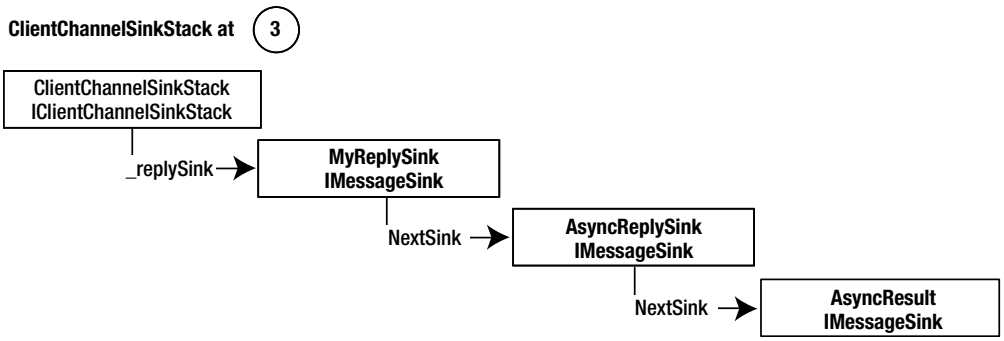

Figure 11-16. *The stack before the call to the SOAP formatter*

The SOAP formatter deserializes the HTTP response and creates a new IMessage object. This object is then passed as a parameter to the channel sink stack's `DispatchReplyMessage()` method.

The sink stack now calls `SyncProcessMessage()` on the first entry of its reply sink chain, which is shown as (4) in the sequence diagram. After this call, the IMessage travels through the sinks until it reaches the AsyncResult object.

This final sink will examine the response message and prepare the return values for the call to the Delegate's `EndInvoke()` method.

Exception Handling

When an exception occurs during an IClientChannelSink's processing, it has to call `DispatchException()` on the sink stack. The stack in this case generates a new ReturnMessage object, passing the original exception as a parameter to its constructor. This newly created return message travels through the chain of IMessageSinks, and the exception will be "unwrapped" by the AsyncResult sink.

Server-Side Asynchronous Processing

On the server side, there are also two kinds of interfaces: IServerChannelSink and IMessageSink. The asynchronous processing for objects implementing IMessageSink is handled in the same way as on the client: the sink has to create another reply sink and pass this to the next sink's `AsyncProcessMessage()` method.

The handling of asynchronous messages for IServerChannelSink objects is a little bit different. You can see the relevant parts of this interface here:

```
public interface IServerChannelSink {
   ServerProcessing ProcessMessage(IServerChannelSinkStack sinkStack,
                                   IMessage requestMsg,
                                   ITransportHeaders requestHeaders,
                                   Stream requestStream,
                                   ref IMessage responseMsg,
                                   ref ITransportHeaders responseHeaders,
                                   ref Stream responseStream);

   void AsyncProcessResponse(IServerResponseChannelSinkStack sinkStack,
                             object state,
                             IMessage msg,
                             ITransportHeaders headers,
                             Stream stream);
}
```

When a serialized message is received by an object implementing this interface, it is not yet determined whether it will be handled synchronously or asynchronously. This can only be defined *after* the message later reaches the formatter.

In the meantime, the sink has to assume that the response *might* be received asynchronously, and therefore will have to push itself (and a possible state object) onto a stack before calling the next sink.

The call to the next sink returns a ServerProcessing value that can be Completed, Async, or OneWay. The sink has to check whether this value is Completed and only in such a case might do any post-processing work in `ProcessMessage()`. If this returned value is OneWay, the sink will not receive any further information when the processing has been finished.

When the next sink's return value is ServerProcessing.Async, the current sink will be notified via the sink stack when the processing has been completed. The sink stack will call `AsyncProcessResponse()` in this case. After the sink has completed the processing of the response, it has to call `sinkStack.AsyncProcessResponse()` to forward the call to further sinks.

A sample implementation of `ProcessMessage()` and `AsyncProcessResponse()` might look like this:

```
public ServerProcessing ProcessMessage(IServerChannelSinkStack sinkStack,
   IMessage requestMsg,
   ITransportHeaders requestHeaders,
   Stream requestStream,
   out IMessage responseMsg,
   out ITransportHeaders responseHeaders,
   out Stream responseStream)
{
```

```
   // handling of the request will be implemented here

   // pushing onto stack and forwarding the call
   sinkStack.Push(this,null);

    ServerProcessing srvProc = _nextSink.ProcessMessage(sinkStack,
      requestMsg,
      requestHeaders,
      requestStream,
      out responseMsg,
      out responseHeaders,
      out responseStream);

   if (srvProc == ServerProcessing.Complete) {
      // handling of the response will be implemented here
   }

   // returning status information
   return srvProc;
}

public void AsyncProcessResponse(IServerResponseChannelSinkStack sinkStack,
   object state,
   IMessage msg,
   ITransportHeaders headers,
   Stream stream)
{

   // handling of the response will be implemented here

   // forwarding to the stack for further processing
   sinkStack.AsyncProcessResponse(msg,headers,stream);
}
```

Summary

In this chapter, you learned about the details and inner workings of .NET Remoting. You read about the various processing stages of a message, and you now know the difference between IMessageSink, IClientChannelSink, and IServerChannelSink. You also know how asynchronous requests are processed, and that the inner workings of the asynchronous message handling is different for message sinks and channel sinks.

In the next chapter, I'll show you how those sinks are created using sink providers.

CHAPTER 12

Creation of Sinks

The previous chapter showed you the various kinds of sinks and their synchronous and asynchronous processing of requests. What I have omitted until now is one of the most important steps: the instantiation of sinks and sink chains. Sinks are normally not created directly in either your code or with the definition in configuration files. Instead, a chain of sink providers is set up, which will in turn return the sink chains on demand. This chapter shows you the foundation on which to build your own sinks. The implementation of those custom sinks is presented in Chapter 13.

Understanding Sink Providers

As you saw in Chapter 4, you can define a chain of sinks in a .NET configuration file as shown in the following code. (This example is for a client-side configuration file; for server-side chains, you have to replace clientProviders with serverProviders.)

```
<configuration>
  <system.runtime.remoting>
    <application>
      <channels>
        <channel ref="http">
          <clientProviders>
             <provider type="MySinks.SomeMessageSinkProvider, Client" />
             <formatter ref="soap" />
             <provider type="MySinks.SomeClientChannelSinkProvider, Client" />
          </clientProviders>
        </channel>
      </channels>
    </application>
  </system.runtime.remoting>
</configuration>
```

To cover this example more thoroughly, I'll expand the `<formatter ref="soap" />` setting using the "real" value from machine.config (including the necessary strong name here).

```
<formatter id="soap",
  type="System.Runtime.Remoting.Channels.SoapClientFormatterSinkProvider,
  System.Runtime.Remoting, Version=1.0.3300.0, Culture=neutral,
  PublicKeyToken=b77a5c561934e089"/>
```

The complete chain now looks like this:

```
<provider type="MySinks.SomeMessageSinkProvider, Client" />
<formatter id="soap",
  type="System.Runtime.Remoting.Channels.SoapClientFormatterSinkProvider,
  System.Runtime.Remoting, Version=1.0.3300.0, Culture=neutral,
  PublicKeyToken=b77a5c561934e089"/>

<provider type="MySinks.SomeClientChannelSinkProvider, Client" />
```

As you can see in these examples, the chain is *not* defined using the sinks' names/types (which would be, for example, SoapClientFormatterSink). Instead, a chain of *providers* is set up. A provider can be either client side or server side and has to implement at least one of these interfaces:

```
public interface IClientChannelSinkProvider
{
    IClientChannelSinkProvider Next { get; set; }

    IClientChannelSink CreateSink(IChannelSender channel,
                                  string url,
                                  object remoteChannelData);
}

public interface IServerChannelSinkProvider
{
    IServerChannelSinkProvider Next { get; set; }

    IServerChannelSink CreateSink(IChannelReceiver channel);
    void GetChannelData(IChannelDataStore channelData);
}
```

You can see in this interface declaration that there is indeed a chain set up using the `Next` property of each provider.

Creating Client-Side Sinks

After loading the configuration file shown previously, this provider chain will consist of the objects shown in Figure 12-1. The first three providers are loaded from the configuration, whereas the last one (HttpClientTransportSinkProvider) is by default instantiated by the HTTP channel.

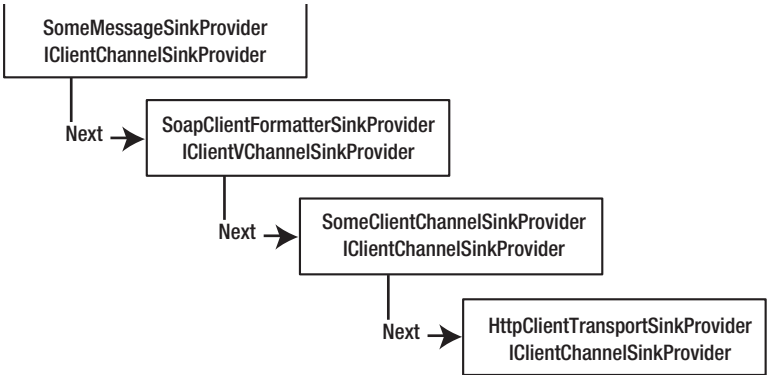

Figure 12-1. *Chain of providers*

On the client side, these sink providers are associated with the client-side channel. You can access the channel object using the following line of code:

```
IChannel chnl = ChannelServices.GetChannel("http");
```

■ **Note** The "http" in this code line refers to the channel's unique name. For HTTP and binary channels, these names are set in machine.config.

The channel object's contents relevant for creation of the sinks are shown in Figure 12-2.

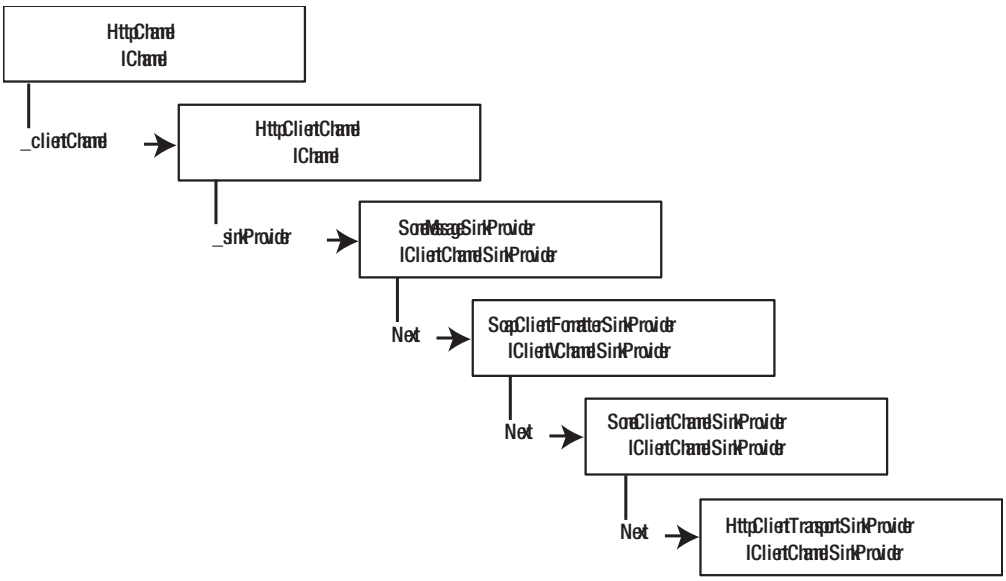

Figure 12-2. *IChannel with populated sink providers*

When a reference to a remote SAO object is created (for CAOs, an additional ConstructionCall message is sent to the server and the proxy's identity object populated), a lot of things happen behind the scenes. At some time during the use of the `new` operator or the call to `Activator.GetObject()`, the method `RemotingServices.Connect()` is called. What happens after this call is shown (in part) in Figure 12-3.

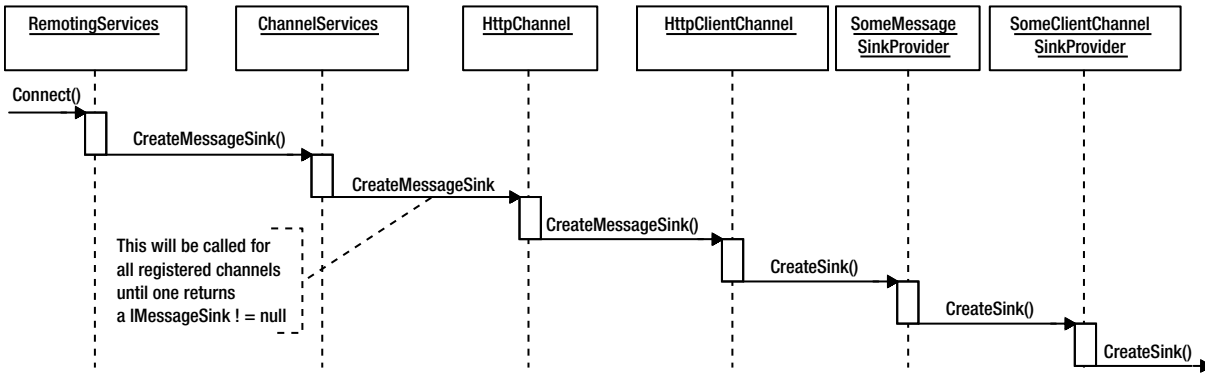

Figure 12-3. *Creation of sinks from a chain of providers*

After being invoked from RemotingServices, ChannelServices calls `CreateMessageSink()` on each registered channel until one of them accepts the URL that is passed as a parameter. The HTTP channel, for example, will work on any URLs that start with http: or https:, whereas the TCP channel will only accept those with a tcp: protocol designator.

When the channel recognizes the given URL, it calls `CreateMessageSink()` on its client-side channel.

■ **Note** The HTTP channel internally consists of both the client-side and the server-side transport channel.

HttpClientChannel, in turn, invokes `CreateSink()` on the first sink provider (as shown in Figure 12-3). What the different sink providers do now corresponds to the following code (shown for a sample SomeClientChannelSinkProvider):

```
public class SomeClientChannelSinkProvider: IClientChannelSinkProvider
{
   private IClientChannelSinkProvider next = null;

   public IClientChannelSink CreateSink(IChannelSender channel,
                                        string url,
                                        object remoteChannelData)
   {
      IClientChannelSink nextSink = null;
```

```
        // checking for additional sink providers
        if (next != null)
      {
         nextSink = next.CreateSink(channel,url,remoteChannelData);
      }

      // returning first entry of a sink chain
      return new SomeClientChannelSink(nextSink);
   }
}
```

Each sink provider first calls `CreateSink()` on the next entry in the provider chain and then returns its own sink on top of the sink chain returned from this call. The exact syntax for placing a new sink at the beginning of the chain is not specified, but in this case the SomeClientSink provides a constructor that takes an IClientChannelSink object as a parameter and sets its `_nextChnlSink` instance variable as shown in the following snippet (again, only parts of the class are shown here):

```
public class SomeClientChannelSink: IClientChannelSink
{
   private IClientChannelSink _nextChnlSink;

   public SomeClientChannelSink (IClientChannelSink next)
   {
      _nextChnlSink = next;
   }
}
```

The complete sink chain that is returned from the call to `ChannelServices.CreateMessageSink()` is then connected to a new TransparentProxy/RealProxy pair's identity object, as shown in Figure 12-4.

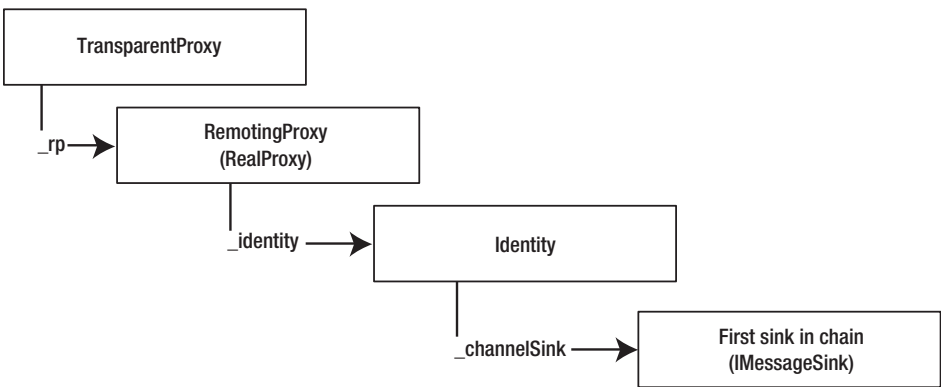

Figure 12-4. *The first IMessageSink is connected to the TransparentProxy.*

Creating Server-Side Sinks

The creation of server-side sinks works a little differently from the creation of the client-side sinks. As you've seen previously, on the client side the necessary sinks are created when a reference to a remote object is acquired. Contrary to this, server-side sinks are created as soon as a channel is registered.

When the server-side channel is created from a definition in a configuration file, the following constructor will be used:

```
public HttpServerChannel(IDictionary properties,
                         IServerChannelSinkProvider sinkProvider)
```

The IDictionary is taken from the attributes of the <channel> section. When your configuration file, for example, contains this line:

```
<channel ref="http" port="1234" />
```

then the properties dictionary will contain one entry with the key "port" and value 1234.

In the `sinkProvider` parameter to the constructor, the first entry to the chain of sink providers will be passed to the channel. This chain is constructed from the entries of the <serverProviders> setting in the configuration file.

During the channel setup, which is started from the HTTP channel's constructor, one of two things will happen now. If the <serverProviders> setting is missing, the default sink chain, which is shown in Figure 12-5, will be created.

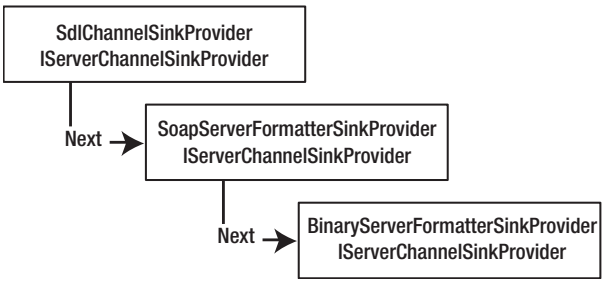

Figure 12-5. *The HttpServerChannel's default sink chain*

When <serverProviders> has been specified in the configuration file, the sink chain will be created from those values, and none of those default sink providers will be used.

■ **Note** This is quite interesting, because in this case, you will not be able to use the ?WSDL parameter to the URL of your SAO to generate WSDL without explicitly specifying SdlChannelSinkProvider in the <serverProviders> section.

After this chain of providers has been created, `ChannelServices.CreateServerChannelSinkChain()` is called. This method takes the sink provider chain as a parameter. It then walks the chain and adds a DispatchChannelSinkProvider object at the end of the chain before calling its `CreateSink()` method. Finally, it returns the generated sink chain. After receiving this object from ChannelServices, HttpServerChannel will add an HttpServerTransportSink as the first element. The resulting server-side channel object is shown in Figure 12-6.

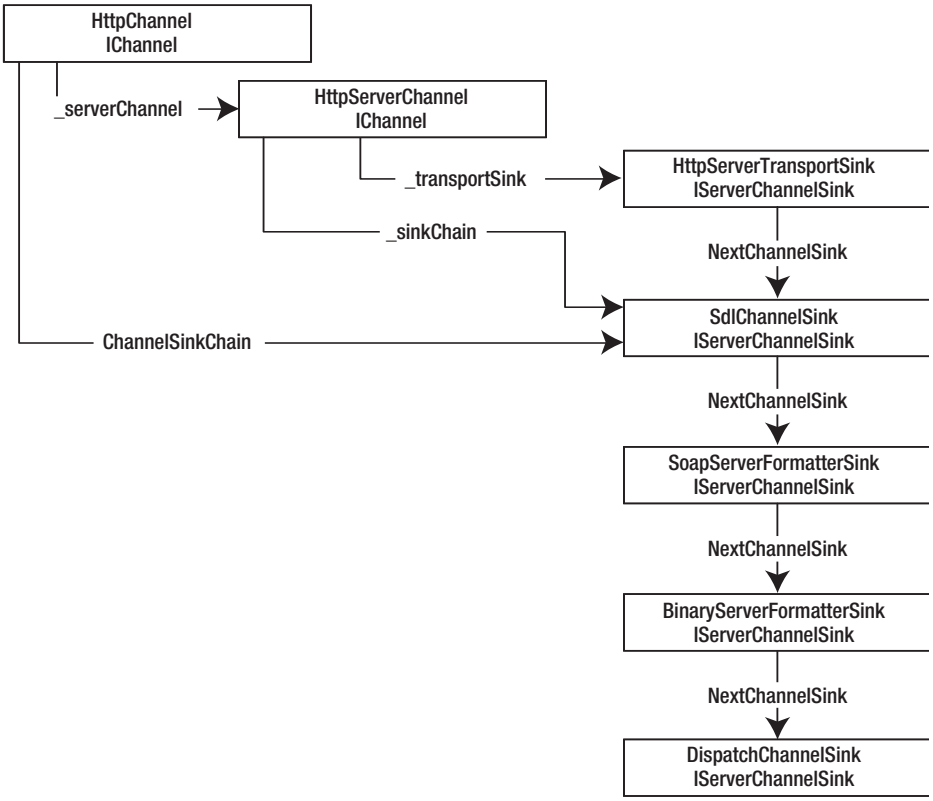

Figure 12-6. *The complete server-side HTTP channel's sink stack*

Using Dynamic Sinks

As you've seen in the previous chapter, both the client-side and the server-side sink chain can call dynamic sinks. On the server side this is done by the CrossContextChannel and on the client side by the ClientContextTerminatorSink.

Dynamic sinks are associated with a specific *context* (you can read more about contexts in Chapter 11) and therefore will be called for *all* calls passing a context boundary. They cannot be assigned to a specific channel and will even be called for local cross-context or cross-AppDomain calls. The sinks are created by dynamic context properties, which are classes implementing IDynamicProperty and IContributeDynamicSink.

■ **Note** IContributeDynamicSink can be compared to a sink provider for dynamic sinks.

The corresponding interfaces are shown here:

```
public interface IDynamicProperty
{
    string Name { get; }
}

public interface IContributeDynamicSink
{
    IDynamicMessageSink GetDynamicSink();
}
```

The dynamic sink itself, which has to be returned from `GetDynamicSink()`, implements the following IDynamicMessageSink interface:

```
public interface IDynamicMessageSink
{
    void ProcessMessageStart(IMessage reqMsg, bool bCliSide, bool bAsync);
    void ProcessMessageFinish(IMessage replyMsg, bool bCliSide, bool bAsync);
}
```

As the names imply, the `ProcessMessageStart()` method is called *before* the message travels further through the chain, and `ProcessMessageFinish()` is called when the call has been handled by the sink.

The following dynamic sink will simply write a line to the console, whenever a message passes a remoting boundary:

```
public class MyDynamicSinkProvider: IDynamicProperty, IContributeDynamicSink
{
   public string Name
   {
      get { return "MyDynamicSinkProvider"; }
   }
```

```
   public IDynamicMessageSink GetDynamicSink()
   {
      return new MyDynamicSink();
   }

}

public class MyDynamicSink: IDynamicMessageSink
{
   public void ProcessMessageStart(IMessage reqMsg, bool bCliSide,
                                    bool bAsync)
   {
      Console.WriteLine("--> MyDynamicSink: ProcessMessageStart");
   }

   public void ProcessMessageFinish(IMessage replyMsg, bool bCliSide,
                                     bool bAsync)
   {
      Console.WriteLine("--> MyDynamicSink: ProcessMessageFinish");
   }
}
```

To register an IDynamicProperty with the current context, you can use the following code:

```
Context ctx = Context.DefaultContext;
IDynamicProperty prp = new MyDynamicSinkProvider();
Context.RegisterDynamicProperty(prp, null, ctx);
```

Summary

In this chapter, you read about the creation of the various kinds of sinks. Together with the previous chapter, you're now fully equipped to implement your own sinks to enhance the feature set of .NET Remoting. You also made first contact with dynamic context sinks, a topic that is covered in more detail in Chapter 15.

I admit that the last two chapters have been quite heavy in content, but in the next chapter I reward your patience by showing some real-world sinks that employ all the techniques presented here.

CHAPTER 13

Extending .NET Remoting

In Chapters 11 and 12, I told you a lot about the various places in your remoting applications that can be extended by custom sinks and providers. What I didn't tell you is *why* you'd want to change the default remoting behavior. There are a lot of reasons for doing so:

- Compression or encryption of the message's contents.
- Passing additional information from the client to the server. For example, you could pass the client-side thread's priority to the server so that the remote execution is performed using the same priority.
- Extending the "look and feel" of .NET Remoting. You could, for example, switch to a per-host authentication model instead of the default per-object model.
- Debugging your applications by dumping the message's contents to the console or to a log file.
- And last but not least, custom sinks and providers enable you to use other transport mediums such as MSMQ or even SMTP/POP3.

You might ask why Microsoft hasn't already implemented these features itself. The only answer I can give you is that most programmers, including myself, really *prefer* being able to change the framework and add the necessary features themselves in a clean and documented way. You can look forward to getting message sinks from various third-party providers, including Web sites from which you can download sinks with included source code.

Creating a Compression Sink

The first thing to ask yourself before starting to implement a new feature for .NET Remoting is whether you'll want to work on the original message (that is, the underlying dictionary) or on the serialized message that you'll get from the stream. In the case of compression, you won't really care about the message's contents and instead just want to compress the resulting stream at the client side and decompress it on the server side *before* reaching the server's SoapFormatter.

You can see the planned client-side sink chain in Figure 13-1 and the server-side chain in Figure 13-2.

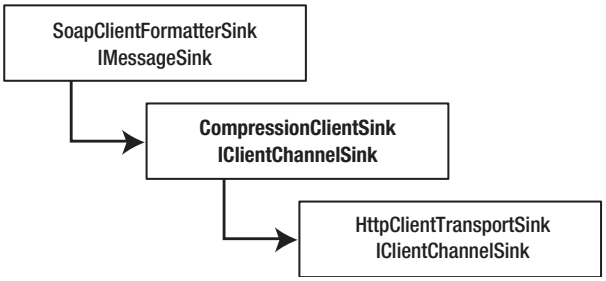

Figure 13-1. *Client-side sink chain with the compression sink*

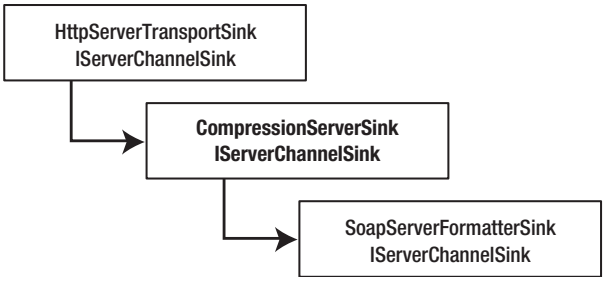

Figure 13-2. *Server-side sink chain with the compression sink*

After having decided upon the two new sinks, you can identify all classes that need to be written.

- *CompressionClientSink*: Implements IClientChannelSink, compresses the request stream, and decompresses the response stream
- *CompressionClientSinkProvider*: Implements IClientChannelSinkProvider and is responsible for the creation of the sink
- *CompressionServerSink*: Implements IServerChannelSink, decompresses the request, and compresses the response before it is sent back to the client
- *CompressionServerSinkProvider*: Implements IServerChannelSinkProvider and creates the server-side sink

Unfortunately, the .NET Framework only added support for compression with version 2.0, so you will have to use a third-party compression library if you are running on version 1.0 or 1.1. I'd like to recommend using Mike Krueger's SharpZipLib (available from `http://www.icsharpcode.net/OpenSource/SharpZipLib/Default.aspx`). This is an open source C# library that is covered by a liberated GPL license. To quote the Web page: "In plain English, this means you can use this library in commercial closed-source applications."

Implementing the Client-Side Sink

The client-side sink extends BaseChannelSinkWithProperties and implements IClientChannelSink. In Listing 13-1, you can see a skeleton client channel sink. The positions at which you'll have to implement the preprocessing and post-processing logic have been marked "TODO." This is a fully working sink—it simply doesn't do anything useful yet.

Listing 13-1. *A Skeleton IClientChannelSink*

```
using System;
using System.Runtime.Remoting.Channels;
using System.Runtime.Remoting.Messaging;
using System.IO;

namespace CompressionSink
{
   public class CompressionClientSink: BaseChannelSinkWithProperties,
                                IClientChannelSink
   {
      private IClientChannelSink _nextSink;

      public CompressionClientSink(IClientChannelSink next)
      {
         _nextSink = next;
      }

      public IClientChannelSink NextChannelSink
      {
         get {
            return _nextSink;
         }
      }


      public void AsyncProcessRequest(IClientChannelSinkStack sinkStack,
                                      IMessage msg,
                                      ITransportHeaders headers,
                                      Stream stream)
      {
         // TODO: Implement the preprocessing

         sinkStack.Push(this,null);
         _nextSink.AsyncProcessRequest(sinkStack,msg,headers,stream);
      }
```

```
      public void AsyncProcessResponse(IClientResponseChannelSinkStack sinkStack,
                                       object state,
                                       ITransportHeaders headers,
                                       Stream stream)
      {

         // TODO: Implement the post-processing

         sinkStack.AsyncProcessResponse(headers,stream);
      }


      public Stream GetRequestStream(IMessage msg,
                                     ITransportHeaders headers)
      {
         return _nextSink.GetRequestStream(msg, headers);
      }


      public void ProcessMessage(IMessage msg,
                                 ITransportHeaders requestHeaders,
                                 Stream requestStream,
                                 out ITransportHeaders responseHeaders,
                                 out Stream responseStream)
      {
         // TODO: Implement the preprocessing

         _nextSink.ProcessMessage(msg,
                                  requestHeaders,
                                  requestStream,
                                  out responseHeaders,
                                  out responseStream);

         // TODO: Implement the post-processing

      }
   }
}
```

Before filling this sink with functionality, you create a helper class that communicates with the compression library and returns a compressed or uncompressed copy of a given stream. You can see this class in Listing 13-2.

Listing 13-2. *Class Returning Compressed or Uncompressed Streams*

```
using System;
using System.IO;
using NZlib.Compression;
using NZlib.Streams;
```

```
namespace CompressionSink {
   public class CompressionHelper {
      public static Stream GetCompressedStreamCopy(Stream inStream) {
         Stream outStream = new System.IO.MemoryStream();
         DeflaterOutputStream compressStream = new DeflaterOutputStream(
            outStream,new Deflater(Deflater.BEST_COMPRESSION));
         byte[] buf = new Byte[1000];
         int cnt = inStream.Read(buf,0,1000);
         while (cnt>0) {
            compressStream.Write(buf,0,cnt);
            cnt = inStream.Read(buf,0,1000);
         }
         compressStream.Finish();
         compressStream.Flush();
         outStream.Seek(0,SeekOrigin.Begin);
         return outStream;
      }

      public static Stream GetUncompressedStreamCopy(Stream inStream) {
         MemoryStream outStream = new MemoryStream();
         inStream = new InflaterInputStream(inStream);
         byte[] buf = new Byte[1000];
         int cnt = inStream.Read(buf,0,1000);
         while (cnt>0)
         {
            outStream.Write(buf,0,cnt);
            cnt = inStream.Read(buf,0,1000);
         }
         outStream.Seek(0,SeekOrigin.Begin);
         return outStream;
      }
   }
}
```

When implementing the compression functionality in the client-side sink, you have to deal with both synchronous and asynchronous processing. The synchronous implementation is quite straightforward. Before passing control further down the chain, the sink simply compresses the stream. When it has finished processing (that is, when the server has sent its response), the message sink will decompress the stream and return it to the calling function as an out parameter.

```
public void ProcessMessage(IMessage msg,
                           ITransportHeaders requestHeaders,
                           Stream requestStream,
                           out ITransportHeaders responseHeaders,
                           out Stream responseStream)
{
   // generate a compressed stream using NZipLib
   requestStream  =  CompressionHelper.GetCompressedStreamCopy(requestStream);
```

```
   // forward the call to the next sink
   _nextSink.ProcessMessage(msg, requestHeaders, requestStream,
                            out responseHeaders, out responseStream);
   // uncompress the response
   responseStream = CompressionHelper.GetUncompressedStreamCopy(responseStream);
}
```

As you've seen in the previous chapter, asynchronous handling is split between two methods. In the current example, you add the compression to `AsyncProcessRequest()` and the decompression to `AsyncProcessResponse()`, as shown in the following piece of code:

```
public void AsyncProcessRequest(IClientChannelSinkStack sinkStack,
                                IMessage msg,
                                ITransportHeaders headers,
                                Stream stream)
{


   // generate a compressed stream using NZipLib
   stream = CompressionHelper.GetCompressedStreamCopy(stream);

   // push onto stack and forward the request
   sinkStack.Push(this,null);
   _nextSink.AsyncProcessRequest(sinkStack,msg,headers,stream);
}


public void AsyncProcessResponse(IClientResponseChannelSinkStack sinkStack,
                                 object state,
                                 ITransportHeaders headers,
                                 Stream stream)
{

   // uncompress the response
   stream = CompressionHelper.GetUncompressedStreamCopy(stream);

   // forward the request
   sinkStack.AsyncProcessResponse(headers,stream);
}
```

Implementing the Server-Side Sink

The server-side sink's task is to decompress the incoming stream before passing it on to the formatter. In Listing 13-3, you can see a skeleton IServerChannelSink.

Listing 13-3. *A Basic IServerChannelSink*

```
using System;
using System.Runtime.Remoting.Channels;
```

```
using System.Runtime.Remoting.Messaging;
using System.IO;

namespace CompressionSink
{

   public class CompressionServerSink: BaseChannelSinkWithProperties,
                                       IServerChannelSink
   {

      private IServerChannelSink _nextSink;

      public CompressionServerSink(IServerChannelSink next)
      {
         _nextSink = next;
      }

      public IServerChannelSink NextChannelSink
      {
         get
         {
            return _nextSink;
         }
      }

      public void AsyncProcessResponse(IServerResponseChannelSinkStack sinkStack,
         object state,
         IMessage msg,
         ITransportHeaders headers,
         Stream stream)
      {
         // TODO: Implement the post-processing

         // forwarding to the stack for further processing
         sinkStack.AsyncProcessResponse(msg,headers,stream);
      }

      public Stream GetResponseStream(IServerResponseChannelSinkStack sinkStack,
         object state,
         IMessage msg,
         ITransportHeaders headers)
      {
         return null;
      }

      public ServerProcessing ProcessMessage(IServerChannelSinkStack sinkStack,
         IMessage requestMsg,
         ITransportHeaders requestHeaders,
```

```
            Stream requestStream,
            out IMessage responseMsg,
            out ITransportHeaders responseHeaders,
            out Stream responseStream)
        {
            // TODO: Implement the preprocessing

            // pushing onto stack and forwarding the call
            sinkStack.Push(this,null);

            ServerProcessing srvProc = _nextSink.ProcessMessage(sinkStack,
               requestMsg,
               requestHeaders,
               requestStream,
               out responseMsg,
               out responseHeaders,
               out responseStream);

            // TODO: Implement the post-processing

            // returning status information
            return srvProc;
        }
    }
}
```

An interesting difference between client-side and server-side sinks is that the server-side sink does not distinguish between synchronous and asynchronous calls during the request stage. Only later in the sink stack will this decision be made and the call possibly returned asynchronously—therefore you always have to push the current sink onto the sinkStack whenever you want the response to be post-processed. To follow the preceding example, you implement `ProcessMessage()` and `AsyncProcessResponse()` to decompress the request and compress the response.

```
public ServerProcessing ProcessMessage(IServerChannelSinkStack sinkStack,
   IMessage requestMsg,
   ITransportHeaders requestHeaders,
   Stream requestStream,
   out IMessage responseMsg,
   out ITransportHeaders responseHeaders,
   out Stream responseStream)
{
   // uncompressing the request
   requestStream =
   CompressionHelper.GetUncompressedStreamCopy(requestStream);

   // pushing onto stack and forwarding the call
   sinkStack.Push(this,null);
```

```
   ServerProcessing srvProc = _nextSink.ProcessMessage(sinkStack,
      requestMsg, requestHeaders, requestStream,
      out responseMsg, out responseHeaders, out responseStream);

   // compressing the response
   responseStream =
   CompressionHelper.GetCompressedStreamCopy(responseStream);

   // returning status information
   return srvProc;
}

public void AsyncProcessResponse(IServerResponseChannelSinkStack sinkStack,
   object state,
   IMessage msg,
   ITransportHeaders headers,
   Stream stream)
{
   // compressing the response
   stream = CompressionHelper.GetCompressedStreamCopy(stream);

   // forwarding to the stack for further processing
   sinkStack.AsyncProcessResponse(msg,headers,stream);
}
```

Congratulations! If you've been following along with the examples, you have now finished your first channel sinks. To start using them, you only have to implement two providers that take care of the sinks' initialization.

Creating the Sink Providers

Before you can use your sinks in a .NET Remoting application, you have to create a server-side and a client-side sink provider. These classes look nearly identical for most sinks you're going to implement.

In the `CreateSink()` method, you first create the next provider's sinks and then put the compression sink on top of the chain before returning it, as shown in Listing 13-4.

Listing 13-4. *The Client-Side Sink Provider*

```
using System;
using System.Runtime.Remoting.Channels;
using System.Collections;

namespace CompressionSink
{
   public class CompressionClientSinkProvider: IClientChannelSinkProvider
   {
      private IClientChannelSinkProvider _nextProvider;
```

```
      public CompressionClientSinkProvider(IDictionary properties,
             ICollection providerData)
      {
         // not yet needed
      }

      public IClientChannelSinkProvider Next
      {
         get {return _nextProvider; }
         set {_nextProvider = value;}
      }

      public IClientChannelSink CreateSink(IChannelSender channel,
           string url,
           object remoteChannelData)
      {
         // create other sinks in the chain
         IClientChannelSink next = _nextProvider.CreateSink(channel,
            url,
            remoteChannelData);

         // put our sink on top of the chain and return it
         return new CompressionClientSink(next);
      }
   }
}
```

The server-side sink provider that is shown in Listing 13-5 looks nearly identical, but returns IServerChannelSink instead of IClientChannelSink.

Listing 13-5. *The Server-Side Sink Provider*

```
using System;
using System.Runtime.Remoting.Channels;
using System.Collections;

namespace CompressionSink
{
   public class CompressionServerSinkProvider: IServerChannelSinkProvider
   {
      private IServerChannelSinkProvider _nextProvider;

      public CompressionServerSinkProvider(IDictionary properties,
             ICollection providerData)
      {
         // not yet needed
      }
```

```
      public IServerChannelSinkProvider Next
      {
         get {return _nextProvider; }
         set {_nextProvider = value;}
      }

      public IServerChannelSink CreateSink(IChannelReceiver channel)
      {
         // create other sinks in the chain
         IServerChannelSink next = _nextProvider.CreateSink(channel);

         // put our sink on top of the chain and return it
         return new CompressionServerSink(next);
      }

      public void GetChannelData(IChannelDataStore channelData)
      {
         // not yet needed
      }

   }
}
```

Using the Sinks

To use the sinks on the client and server side of a channel, you simply have to include them in your configuration files. In the client-side configuration file, you have to incorporate the information shown in the following code. If you place the CompressionSink assembly in the GAC, mind that you have to specify the complete strong name in the type attribute!

```
<configuration>
 <system.runtime.remoting>
  <application>
   <channels>
    <channel ref="http">

      <clientProviders>
         <formatter ref="soap" />
         <provider
         type="CompressionSink.CompressionClientSinkProvider, CompressionSink" />
       </clientProviders>

    </channel>
   </channels>
  </application>
 </system.runtime.remoting>
</configuration>
```

The server-side configuration file will look similar to the following:

```
<configuration>
 <system.runtime.remoting>
  <application>
   <channels>
    <channel ref="http" port="1234">

     <serverProviders>
         <provider
         type="CompressionSink.CompressionServerSinkProvider, CompressionSink" />
         <formatter ref="soap"/>
      </serverProviders>

    </channel>
   </channels>
  </application>
 </system.runtime.remoting>
</configuration>
```

Figure 13-3 shows a TCP trace from a client/server connection that isn't using this sink, whereas Figure 13-4 shows the improvement when compression is used.

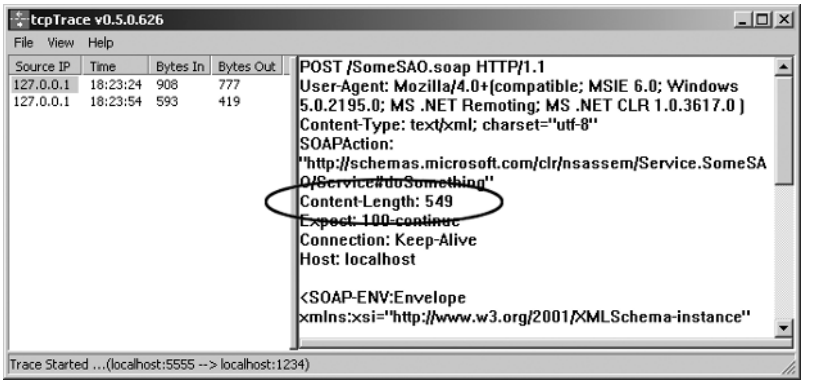

Figure 13-3. *TCP trace of an HTTP/SOAP connection*[1]

1. You can get this tcpTrace tool at `http://www.pocketsoap.com`.

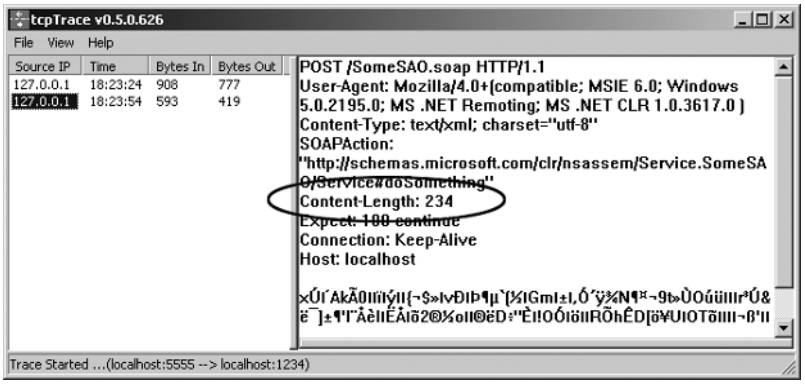

Figure 13-4. *TCP trace of an HTTP connection with compressed content*

In the circled area, you can see that the HTTP Content-Length header goes down from 549 bytes to 234 bytes when using the compression sink.

■ **Note** This is a *proof-of-concept* example. Instead of using compression in this scenario, you could easily switch to binary encoding to save even more bytes to transfer. But keep in mind that the compression sink also works with the binary formatter!

Extending the Compression Sink

The server-side sink as presented in the previous section has at least one serious problem when used in real-world applications: it doesn't yet detect whether the stream is compressed or not and will always try to decompress it. This will lead to an inevitable exception when the request stream has not been compressed before.

In an average remoting scenario, you have two types of users. On the one hand, there are local (LAN) users who connect to the server via high-speed links. If these users compress their requests, it's quite possible that the stream compression would take up more time (in regard to client- and server-side CPU time plus transfer time) than the network transfer of the uncompressed stream would. On the other hand, you might have several remote users who connect via lines ranging from speedy T1s down to 9600 bps wireless devices. *These* users will quite certainly profit from sending requests in a compressed way.

The first step to take when implementing these additional capabilities in a channel sink is to determine how the server will know that the request stream is compressed. Generally this can be done by adding additional fields to the ITransportHeader object that is passed as a parameter to `ProcessMessage()` and `AsyncProcessRequest()`.

These headers are then transferred to the server and can be obtained by the server-side sink by using the ITransportHeaders that it receives as a parameter to its `ProcessMessage()` method. By convention, these additional headers should start with the prefix X-, so that you can simply add a statement like the following in the client-side sink's `ProcessMessage()` method to indicate that the content will be compressed:

```
requestHeaders["X-Compress"]="yes";
```

The complete `AsyncProcessRequest()` method now looks like this:

```
public void AsyncProcessRequest(IClientChannelSinkStack sinkStack,
                        IMessage msg,
                        ITransportHeaders headers,
                        Stream stream)
{
   headers["X-Compress"]="yes";

   stream = CompressionHelper.GetCompressedStreamCopy(stream);

   // push onto stack and forward the request
   sinkStack.Push(this,null);
   _nextSink.AsyncProcessRequest(sinkStack,msg,headers,stream);
}
```

When the server receives this request, it processes the message and replies with a compressed stream as well. The server also indicates this compression by setting the X-Compress header. The complete client-side code for `AsyncProcessResponse()` and `ProcessMessage()` will therefore look at the response headers and decompress the message if necessary.

```
public void ProcessMessage(IMessage msg,
                    ITransportHeaders requestHeaders,
                    Stream requestStream,
                    out ITransportHeaders responseHeaders,
                    out Stream responseStream)
{

   requestStream = CompressionHelper.GetCompressedStreamCopy(requestStream);
   requestHeaders["X-Compress"] = "yes";

   // forward the call to the next sink
   _nextSink.ProcessMessage(msg,
                        requestHeaders,
                        requestStream,
                        out responseHeaders,
                        out responseStream);

   // deflate the response if necessary
   String xcompress = (String) responseHeaders["X-Compress"];

   if (xcompress != null && xcompress == "yes")
   {
      responseStream =
            CompressionHelper.GetUncompressedStreamCopy(responseStream);
   }
}
```

```
public void AsyncProcessResponse(IClientResponseChannelSinkStack sinkStack,
                           object state,
                           ITransportHeaders headers,
                           Stream stream)
{

   // decompress the stream if necessary
   String xcompress = (String) headers["X-Compress"];

   if (xcompress != null && xcompress == "yes")
   {
      stream = CompressionHelper.GetUncompressedStreamCopy(stream);
   }


   // forward the request
   sinkStack.AsyncProcessResponse(headers,stream);
}
```

The server-side channel sink's `ProcessMessage()` method works a little bit differently. As you've seen in Chapter 11, when the message reaches this method, it's not yet determined whether the call will be executed synchronously or asynchronously. Therefore the sink has to push itself onto a sink stack that will be used when replying asynchronously.

As the AsyncProcessResponse() method for the channel sink has to know whether the original request has been compressed or not, you'll need to use the second parameter of the sinkStack.Push() method, which is called during ProcessMessage(). In this parameter you can put any object that enables you to later determine the state of the request. This state object will be passed as a parameter to `AsyncProcessResponse()`. The complete server-side implementation of `ProcessMessage()` and `AsyncProcessResponse()` therefore looks like this:

```
public ServerProcessing ProcessMessage(IServerChannelSinkStack sinkStack,
   IMessage requestMsg,
   ITransportHeaders requestHeaders,
   Stream requestStream,
   out IMessage responseMsg,
   out ITransportHeaders responseHeaders,
   out Stream responseStream)
{

   bool isCompressed=false;


   // decompress the stream if necessary
   String xcompress = (String) requestHeaders["X-Compress"];
   if (xcompress != null && xcompress == "yes")
   {
      requestStream = CompressionHelper.GetUncompressedStreamCopy(requestStream);
      isCompressed = true;
   }
```

```
        // pushing onto stack and forwarding the call.
        // the state object contains true if the request has been compressed,
        // else false.
        sinkStack.Push(this,isCompressed);

        ServerProcessing srvProc = _nextSink.ProcessMessage(sinkStack,
            requestMsg,
            requestHeaders,
            requestStream,
            out responseMsg,
            out responseHeaders,
            out responseStream);

        if (srvProc == ServerProcessing.Complete ) {
            // compressing the response if necessary
            if (isCompressed)
            {
                responseStream=
                       CompressionHelper.GetCompressedStreamCopy(responseStream);
                responseHeaders["X-Compress"] = "yes";
            }
        }
        // returning status information
        return srvProc;
    }

    public void AsyncProcessResponse(IServerResponseChannelSinkStack sinkStack,
        object state,
        IMessage msg,
        ITransportHeaders headers,
        Stream stream)
    {
        // fetching the flag from the async-state
        bool hasBeenCompressed = (bool) state;

        // compressing the response if necessary
        if (hasBeenCompressed)
        {
            stream=CompressionHelper.GetCompressedStreamCopy(stream);
            headers["X-Compress"] = "yes";
        }


        // forwarding to the stack for further processing
        sinkStack.AsyncProcessResponse(msg,headers,stream);
    }
```

As you can see in Figure 13-5, which shows the HTTP request, and Figure 13-6, which shows the corresponding response, the complete transfer is compressed and the custom HTTP header X-Compress is populated.

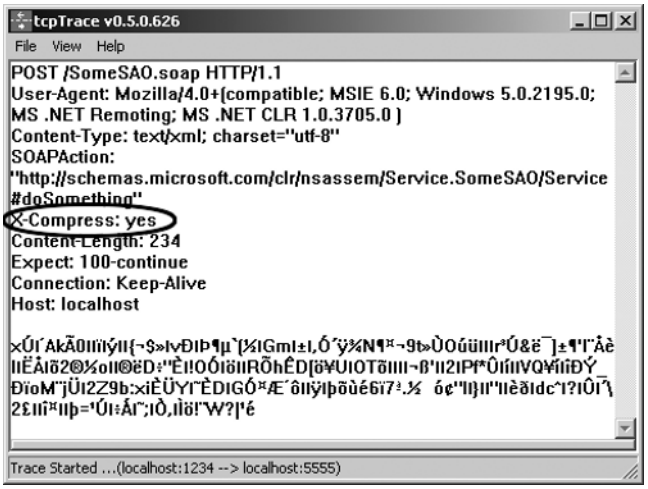

Figure 13-5. *The compressed HTTP request*

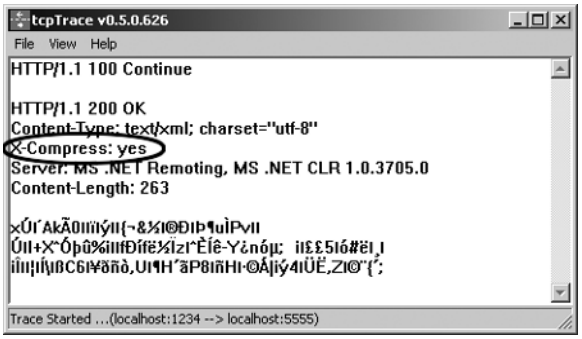

Figure 13-6. *The compressed HTTP response*

Encrypting the Transfer

Even though using an asymmetric/symmetric combination such as HTTPS/SSL for the encryption of the network traffic provides the only real security, in some situations HTTPS isn't quite helpful.

First, .NET Remoting by default only supports encryption when using an HTTP channel and when hosting the server-side components in IIS. If you want to use a TCP channel or host your objects in a Windows service, there's no default means of secure communication.

Second, even if you use IIS to host your components, callbacks that are employed with event notification will *not* be secured. This is because your client (which is the server for the callback object) does not publish its objects using HTTPS, but only HTTP.

Essential Symmetric Encryption

Symmetric encryption is based on one key fact: client and server will have access to the *same* encryption key. This key is not a password as you might know it, but instead is a binary array in common sizes from 40 to 192 bits. Additionally, you have to choose from among a range of encryption algorithms supplied with the .NET Framework: DES, TripleDES, RC2, or Rijndael.

To generate a random key for a specified algorithm, you can use the following code snippet. You will find the key in the `byte[]` variable `mykey` afterwards.

```
String algorithmName = "TripleDES";
SymmetricAlgorithm alg = SymmetricAlgorithm.Create(algorithmName);

int keylen = 128;
alg.KeySize = keylen;
alg.GenerateKey();

byte[] mykey = alg.Key;
```

Because each algorithm has a limited choice of valid key lengths, and because you might want to save this key to a file, you can run the separate KeyGenerator console application, which is shown in Listing 13-6.

Listing 13-6. *A Complete Keyfile Generator*

```
using System;
using System.IO;
using System.Security.Cryptography;

class KeyGen
{
   static void Main(string[] args)
   {
      if (args.Length != 1 && args.Length != 3)
      {
         Console.WriteLine("Usage:");
         Console.WriteLine("KeyGenerator <Algorithm> [<KeySize> <Outputfile>]");
         Console.WriteLine("Algorithm can be: DES, TripleDES, RC2 or Rijndael");
         Console.WriteLine();
         Console.WriteLine("When only <Algorithm> is specified, the program");
         Console.WriteLine("will print a list of valid key sizes.");
         return;
      }

      String algorithmname = args[0];
```

```
SymmetricAlgorithm alg = SymmetricAlgorithm.Create(algorithmname);

if (alg == null)
{
   Console.WriteLine("Invalid algorithm specified.");
   return;
}

if (args.Length == 1)
{
   // just list the possible key sizes
   Console.WriteLine("Legal key sizes for algorithm {0}:",algorithmname);
   foreach (KeySizes size in alg.LegalKeySizes)
   {
      if (size.SkipSize != 0)
      {
         for (int i = size.MinSize;i<=size.MaxSize;i=i+size.SkipSize)
         {
            Console.WriteLine("{0} bit", i);
         }
      }
      else
      {
         if (size.MinSize != size.MaxSize)
         {
            Console.WriteLine("{0} bit", size.MinSize);
            Console.WriteLine("{0} bit", size.MaxSize);
         }
         else
         {
            Console.WriteLine("{0} bit", size.MinSize);
         }
      }
   }
   return;
}

// user wants to generate a key
int keylen = Convert.ToInt32(args[1]);
String outfile = args[2];
try
{
   alg.KeySize = keylen;
   alg.GenerateKey();
   FileStream fs = new FileStream(outfile,FileMode.CreateNew);
   fs.Write(alg.Key,0,alg.Key.Length);
   fs.Close();
```

```
            Console.WriteLine("{0} bit key written to {1}.",
                        alg.Key.Length * 8,
                        outfile);

         }
         catch (Exception e)
         {
            Console.WriteLine("Exception: {0}" ,e.Message);
            return;
         }


      }
}
```

When this key generator is invoked with KeyGenerator.exe (without any parameters), it will print a list of possible algorithms. You can then run KeyGenerator.exe <AlgorithmName> to get a list of possible key sizes for the chosen algorithm. To finally generate the key, you have to start KeyGenerator.exe <AlgorithmName> <KeySize> <OutputFile>. To generate a 128-bit key for a TripleDES algorithm and save it in c:\testfile.key, run KeyGenerator.exe TripleDES 128 c:\testfile.key.

The Initialization Vector

Another basic of symmetric encryption is the use of a random *initialization vector* (IV). This is again a byte array, but it's not statically computed during the application's development. Instead, a new one is generated for each encryption taking place.

To successfully decrypt the message, both the key and the initialization vector have to be known to the second party. The key is determined during the application's deployment (at least in the following example) and the IV has to be sent via remoting boundaries with the original message. The IV is therefore not secret on its own.

Creating the Encryption Helper

Next I show you how to build this sink in the same manner as the previous CompressionSink, which means that the sink's core logic will be extracted to a helper class. I call this class EncryptionHelper. The encryption helper will implement two methods, `ProcessOutboundStream()` and `ProcessInboundStream()`. The methods' signatures look like this:

```
public static Stream ProcessOutboundStream(
         Stream inStream,
         String algorithm,
         byte[] encryptionkey,
         out byte[] encryptionIV)

public static Stream ProcessInboundStream(
         Stream inStream,
         String algorithm,
         byte[] encryptionkey,
         byte[] encryptionIV)
```

As you can see in the signatures, both methods take a stream, the name of a valid crypto-algorithm, and a byte array that contains the encryption key as parameters. The first method is used to encrypt the stream. It also internally generates the IV and returns it as an out parameter. This IV then has to be serialized by the sink and passed to the other party in the remoting call. `ProcessInboundStream()`, on the other hand, expects the IV to be passed to it, so this value has to be obtained by the sink before calling this method. The implementation of these helper methods can be seen in Listing 13-7.

Listing 13-7. *The EncryptionHelper Encapsulates the Details of the Cryptographic Process*

```
using System;
using System.IO;
using System.Security.Cryptography;

namespace EncryptionSink
{

   public class EncryptionHelper
   {

      public static Stream ProcessOutboundStream(
         Stream inStream,
         String algorithm,
         byte[] encryptionkey,
         out byte[] encryptionIV)
      {
         Stream outStream = new System.IO.MemoryStream();

         // set up the encryption properties
         SymmetricAlgorithm alg = SymmetricAlgorithm.Create(algorithm);
         alg.Key = encryptionkey;
         alg.GenerateIV();
         encryptionIV = alg.IV;

         CryptoStream encryptStream = new CryptoStream(
            outStream,
            alg.CreateEncryptor(),
            CryptoStreamMode.Write);

         // write the whole contents through the new streams
         byte[] buf = new Byte[1000];
         int cnt = inStream.Read(buf,0,1000);
         while (cnt>0)
         {
            encryptStream.Write(buf,0,cnt);
            cnt = inStream.Read(buf,0,1000);
         }
         encryptStream.FlushFinalBlock();
         outStream.Seek(0,SeekOrigin.Begin);
```

```
            return outStream;
        }

        public static Stream ProcessInboundStream(
              Stream inStream,
              String algorithm,
              byte[] encryptionkey,
              byte[] encryptionIV)
        {
            // set up decryption properties
            SymmetricAlgorithm alg = SymmetricAlgorithm.Create(algorithm);
            alg.Key = encryptionkey;
            alg.IV = encryptionIV;

            // add the decryptor layer to the stream
            Stream outStream = new CryptoStream(inStream,
               alg.CreateDecryptor(),
               CryptoStreamMode.Read);

            return outStream;
        }

    }
}
```

Creating the Sinks

The EncryptionClientSink and EncryptionServerSink look quite similar to the previous compression sinks. The major difference is that they have custom constructors that are called from their sink providers to set the specified encryption algorithm and key. For outgoing requests, the sinks will set the X-Encrypt header to "yes" and store the initialization vector in Base64 coding in the X-EncryptIV header. The complete client-side sink is shown in Listing 13-8.

Listing 13-8. *The EncryptionClientSink*

```
using System;
using System.Runtime.Remoting.Channels;
using System.Runtime.Remoting.Messaging;
using System.IO;
using System.Text;

namespace EncryptionSink
{
    public class EncryptionClientSink: BaseChannelSinkWithProperties,
                                IClientChannelSink
    {
        private IClientChannelSink _nextSink;
        private byte[] _encryptionKey;
        private String _encryptionAlgorithm;
```

```
public EncryptionClientSink(IClientChannelSink next,
   byte[] encryptionKey,
   String encryptionAlgorithm)
{
   _encryptionKey = encryptionKey;
   _encryptionAlgorithm = encryptionAlgorithm;
   _nextSink = next;
}

public void ProcessMessage(IMessage msg,
   ITransportHeaders requestHeaders,
   Stream requestStream,
   out ITransportHeaders responseHeaders,
   out Stream responseStream)
{

   byte[] IV;

   requestStream = EncryptionHelper.ProcessOutboundStream(requestStream,
      _encryptionAlgorithm,_encryptionKey,out IV);

   requestHeaders["X-Encrypt"]="yes";
   requestHeaders["X-EncryptIV"]= Convert.ToBase64String(IV);


   // forward the call to the next sink
   _nextSink.ProcessMessage(msg,
      requestHeaders,
      requestStream,
      out responseHeaders,
      out responseStream);


   if (responseHeaders["X-Encrypt"] != null &&
      responseHeaders["X-Encrypt"].Equals("yes"))
   {

      IV = Convert.FromBase64String(
              (String) responseHeaders["X-EncryptIV"]);

      responseStream = EncryptionHelper.ProcessInboundStream(
         responseStream,
         _encryptionAlgorithm,
         _encryptionKey,
         IV);
   }

}
```

```
public void AsyncProcessRequest(IClientChannelSinkStack sinkStack,
                        IMessage msg,
                        ITransportHeaders headers,
                        Stream stream)
{
   byte[] IV;

   stream = EncryptionHelper.ProcessOutboundStream(stream,
      _encryptionAlgorithm,_encryptionKey,out IV);

   headers["X-Encrypt"]="yes";
   headers["X-EncryptIV"]= Convert.ToBase64String(IV);

   // push onto stack and forward the request
   sinkStack.Push(this,null);
   _nextSink.AsyncProcessRequest(sinkStack,msg,headers,stream);
}


public void AsyncProcessResponse(IClientResponseChannelSinkStack sinkStack,
                           object state,
                           ITransportHeaders headers,
                           Stream stream)
{


   if (headers["X-Encrypt"] != null && headers["X-Encrypt"].Equals("yes"))
   {

      byte[] IV =
         Convert.FromBase64String((String) headers["X-EncryptIV"]);

      stream = EncryptionHelper.ProcessInboundStream(
         stream,
         _encryptionAlgorithm,
         _encryptionKey,
         IV);
   }

   // forward the request
   sinkStack.AsyncProcessResponse(headers,stream);
}


public Stream GetRequestStream(IMessage msg,
                               ITransportHeaders headers)
{
   return null; // request stream will be manipulated later
}
```

```
      public IClientChannelSink NextChannelSink {
         get
         {
            return _nextSink;
         }
      }

   }
}
```

The EncryptionServerSink shown in Listing 13-9 works basically in the same way as the CompressionServerSink does. It first checks the headers to determine whether the request has been encrypted. If this is the case, it retrieves the encryption initialization vector from the header and calls EncryptionHelper to decrypt the stream.

Listing 13-9. *The EncryptionServerSink*

```
using System;
using System.Runtime.Remoting.Channels;
using System.Runtime.Remoting;
using System.Runtime.Remoting.Messaging;
using System.IO;

namespace EncryptionSink
{

   public class EncryptionServerSink: BaseChannelSinkWithProperties,
                              IServerChannelSink
   {

      private IServerChannelSink _nextSink;
      private byte[] _encryptionKey;
      private String _encryptionAlgorithm;

      public EncryptionServerSink(IServerChannelSink next, byte[] encryptionKey,
            String encryptionAlgorithm)
      {
         _encryptionKey = encryptionKey;
         _encryptionAlgorithm = encryptionAlgorithm;
         _nextSink = next;
      }

      public ServerProcessing ProcessMessage(IServerChannelSinkStack sinkStack,
         IMessage requestMsg,
         ITransportHeaders requestHeaders,
         Stream requestStream,
         out IMessage responseMsg,
         out ITransportHeaders responseHeaders,
         out Stream responseStream) {
```

```
bool isEncrypted=false;

// checking the headers
if (requestHeaders["X-Encrypt"] != null &&
   requestHeaders["X-Encrypt"].Equals("yes"))
{
   isEncrypted = true;

   byte[] IV = Convert.FromBase64String(
      (String) requestHeaders["X-EncryptIV"]);

   // decrypt the request
   requestStream = EncryptionHelper.ProcessInboundStream(
      requestStream,
      _encryptionAlgorithm,
      _encryptionKey,
      IV);
}


// pushing onto stack and forwarding the call,
// the flag "isEncrypted" will be used as state
sinkStack.Push(this,isEncrypted);

ServerProcessing srvProc = _nextSink.ProcessMessage(sinkStack,
   requestMsg,
   requestHeaders,
   requestStream,
   out responseMsg,
   out responseHeaders,
   out responseStream);

if (isEncrypted)
{
   // encrypting the response if necessary
   byte[] IV;

   responseStream =
       EncryptionHelper.ProcessOutboundStream(responseStream,
      _encryptionAlgorithm,_encryptionKey,out IV);

   responseHeaders["X-Encrypt"]="yes";
   responseHeaders["X-EncryptIV"]= Convert.ToBase64String(IV);
}
```

```
            // returning status information
            return srvProc;
        }

        public void AsyncProcessResponse(IServerResponseChannelSinkStack sinkStack,
            object state,
            IMessage msg,
            ITransportHeaders headers,
            Stream stream)
        {
            // fetching the flag from the async-state
            bool isEncrypted = (bool) state;


            if (isEncrypted)
            {
                // encrypting the response if necessary
                byte[] IV;

                stream = EncryptionHelper.ProcessOutboundStream(stream,
                    _encryptionAlgorithm,_encryptionKey,out IV);

                headers["X-Encrypt"]="yes";
                headers["X-EncryptIV"]= Convert.ToBase64String(IV);
            }


            // forwarding to the stack for further processing
            sinkStack.AsyncProcessResponse(msg,headers,stream);
        }

        public Stream GetResponseStream(IServerResponseChannelSinkStack sinkStack,
            object state,
            IMessage msg,
            ITransportHeaders headers)
        {
            return null;
        }

        public IServerChannelSink NextChannelSink {
            get {
                return _nextSink;
            }
        }
    }
}
```

Creating the Providers

Contrary to the previous sink, the EncryptionSink expects certain parameters to be present in the configuration file. The first one is "algorithm", which specifies the cryptographic algorithm that should be used (DES, TripleDES, RC2, or Rijndael). The second parameter, "keyfile", specifies the location of the previously generated symmetric keyfile. The same file has to be available to both the client and the server sink.

The following excerpt from a configuration file shows you how the client-side sink will be configured:

```
<configuration>
 <system.runtime.remoting>
  <application>
   <channels>
    <channel ref="http">

    <clientProviders>
     <formatter ref="soap" />
     <provider type="EncryptionSink.EncryptionClientSinkProvider, EncryptionSink"
                         algorithm="TripleDES" keyfile="testkey.dat" />
     </clientProviders>

    </channel>
   </channels>
  </application>
 </system.runtime.remoting>
</configuration>
```

In the following snippet you see how the server-side sink can be initialized:

```
<configuration>
 <system.runtime.remoting>
  <application>
   <channels>
    <channel ref="http" port="5555">

     <serverProviders>
<provider type="EncryptionSink.EncryptionClientSinkProvider, EncryptionSink"
                         algorithm="TripleDES" keyfile="testkey.dat" />
     <formatter ref="soap"/>
     </serverProviders>

    </channel>
   </channels>
  </application>
 </system.runtime.remoting>
</configuration>
```

You can access additional parameters in the sink provider's constructor, as shown in the following source code fragment:

```
public EncryptionClientSinkProvider(IDictionary properties,
       ICollection providerData)
{
  String encryptionAlgorithm = (String) properties["algorithm"];
}
```

In addition to reading the relevant configuration file parameters, both the client-side sink provider (shown in Listing 13-10) and the server-side sink provider (shown in Listing 13-11) have to read the specified keyfile and store it in a byte array. The encryption algorithm and the encryption key are then passed to the sink's constructor.

Listing 13-10. *The EncryptionClientSinkProvider*

```
using System;
using System.IO;
using System.Runtime.Remoting.Channels;
using System.Runtime.Remoting;
using System.Collections;

namespace EncryptionSink
{
   public class EncryptionClientSinkProvider: IClientChannelSinkProvider
   {
      private IClientChannelSinkProvider _nextProvider;

      private byte[] _encryptionKey;
      private String _encryptionAlgorithm;

      public EncryptionClientSinkProvider(IDictionary properties,
               ICollection providerData)
      {
         _encryptionAlgorithm = (String) properties["algorithm"];
         String keyfile = (String) properties["keyfile"];

         if (_encryptionAlgorithm == null || keyfile == null)
         {
            throw new RemotingException("'algorithm' and 'keyfile' have to " +
               "be specified for EncryptionClientSinkProvider");
         }


         // read the encryption key from the specified file
         FileInfo fi = new FileInfo(keyfile);

         if (!fi.Exists)
         {
```

```
            throw new RemotingException("Specified keyfile does not exist");
         }

         FileStream fs = new FileStream(keyfile,FileMode.Open);
         _encryptionKey = new Byte[fi.Length];
         fs.Read(_encryptionKey,0,_encryptionKey.Length);
      }

      public IClientChannelSinkProvider Next
      {
         get {return _nextProvider; }
         set {_nextProvider = value;}
      }

      public IClientChannelSink CreateSink(IChannelSender channel, string url,
             object remoteChannelData)
      {
         // create other sinks in the chain
         IClientChannelSink next = _nextProvider.CreateSink(channel,
            url, remoteChannelData);

         // put our sink on top of the chain and return it
         return new EncryptionClientSink(next,_encryptionKey,
            _encryptionAlgorithm);
      }
   }
}
```

Listing 13-11. *The EncryptionServerSinkProvider*

```
using System;
using System.IO;
using System.Runtime.Remoting.Channels;
using System.Runtime.Remoting;
using System.Collections;

namespace EncryptionSink
{
   public class EncryptionServerSinkProvider: IServerChannelSinkProvider
   {
      private byte[] _encryptionKey;
      private String _encryptionAlgorithm;

      private IServerChannelSinkProvider _nextProvider;

      public EncryptionServerSinkProvider(IDictionary properties,
          ICollection providerData)
      {
```

```
            _encryptionAlgorithm = (String) properties["algorithm"];
            String keyfile = (String) properties["keyfile"];

            if (_encryptionAlgorithm == null || keyfile == null)
            {
               throw new RemotingException("'algorithm' and 'keyfile' have to " +
                  "be specified for EncryptionServerSinkProvider");
            }


            // read the encryption key from the specified file
            FileInfo fi = new FileInfo(keyfile);

            if (!fi.Exists)
            {
               throw new RemotingException("Specified keyfile does not exist");
            }

            FileStream fs = new FileStream(keyfile,FileMode.Open);
            _encryptionKey = new Byte[fi.Length];
            fs.Read(_encryptionKey,0,_encryptionKey.Length);
         }

         public IServerChannelSinkProvider Next
         {
            get {return _nextProvider; }
            set {_nextProvider = value;}
         }

         public IServerChannelSink CreateSink(IChannelReceiver channel)
         {
            // create other sinks in the chain
            IServerChannelSink next = _nextProvider.CreateSink(channel);

            // put our sink on top of the chain and return it
            return new EncryptionServerSink(next,
               _encryptionKey,_encryptionAlgorithm);
         }

         public void GetChannelData(IChannelDataStore channelData)
         {
            // not yet needed
         }

      }
   }
```

When including the sink providers in your configuration files as presented previously, the transfer will be encrypted, as shown in Figure 13-7.

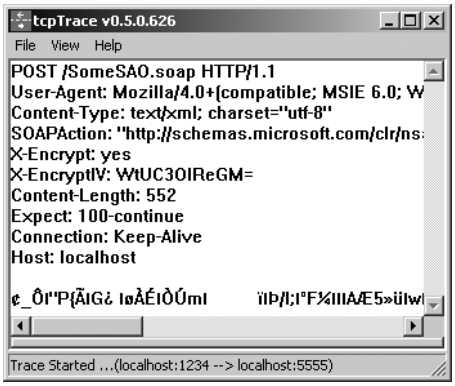

Figure 13-7. *A TCP-trace of the encrypted HTTP traffic*

You can, of course, also chain the encryption and compression sinks together to receive an encrypted *and* compressed stream.

Passing Runtime Information

The previous sinks were IClientChannelSinks and IServerChannelSinks. This means that they work on the resulting stream *after* the formatter has serialized the IMessage object. IMessageSinks, in contrast, can work directly on the message's contents *before* they are formatted. This means that any changes you make to the IMessage's contents will be serialized and therefore reflected in the resulting stream.

■ **Caution** Even though you might be tempted to change the IMessage object's content in an IClientChannelSink, be aware that this change is *not* propagated to the server, because the serialized stream has already been generated from the underlying IMessage!

Because of this distinction, client-side IMessageSinks can be used to pass runtime information from the client to the server. In the following example, I show you how to send the client-side thread's current priority to the server so that remote method calls will execute with the same priority.

To send arbitrary data from the client to the server, you can put it into the Message object's logical call context. In this way, you can transfer objects that either are serializable or extend MarshalByRefObject. For example, to pass the client-side thread's current context for every method call to the server, you can implement the following `SyncProcessMessage()` method:

```
public IMessage SyncProcessMessage(IMessage msg)
{
   if (msg as IMethodCallMessage != null)
   {
      LogicalCallContext lcc =
         (LogicalCallContext) msg.Properties["__CallContext"];

      lcc.SetData("priority",Thread.CurrentThread.Priority);
      return _nextMsgSink.SyncProcessMessage(msg);
   }
   else
   {
      return _nextMsgSink.SyncProcessMessage(msg);
   }
}
```

The same has to be done for `AsyncProcessMessage()` as well.

```
public IMessageCtrl AsyncProcessMessage(IMessage msg, IMessageSink replySink)
{
   if (msg as IMethodCallMessage != null)
   {
      LogicalCallContext lcc =
         (LogicalCallContext) msg.Properties["__CallContext"];

      lcc.SetData("priority",Thread.CurrentThread.Priority);
      return _nextMsgSink.AsyncProcessMessage(msg,replySink);
   }
   else
   {
      return _nextMsgSink.AsyncProcessMessage(msg,replySink);
   }
}
```

On the server side, you have to implement an IServerChannelSink to take the call context from the IMessage object and set `Thread.CurrentThread.Priority` to the contained value.

```
public ServerProcessing ProcessMessage(IServerChannelSinkStack sinkStack,
   IMessage requestMsg,
   ITransportHeaders requestHeaders,
   Stream requestStream,
   out IMessage responseMsg,
   out ITransportHeaders responseHeaders,
   out Stream responseStream)
{
   LogicalCallContext lcc =
      (LogicalCallContext) requestMsg.Properties["__CallContext"];

   // storing the current priority
   ThreadPriority oldprio = Thread.CurrentThread.Priority;
```

```
   // check if the logical call context contains "priority"
   if (lcc != null && lcc.GetData("priority") != null)
   {
      // fetch the priority from the call context
      ThreadPriority priority =
         (ThreadPriority) lcc.GetData("priority");

      Console.WriteLine("  -> Pre-execution priority change {0} to {1}",
         oldprio.ToString(),priority.ToString());

      // set the priority
      Thread.CurrentThread.Priority = priority;
   }



   // push on the stack and pass the call to the next sink
   // the old priority will be used as "state" for the response
   sinkStack.Push(this,oldprio);

   ServerProcessing spres =  _next.ProcessMessage (sinkStack,
      requestMsg, requestHeaders, requestStream,
      out responseMsg,out responseHeaders,out responseStream);

   // restore priority if call is not asynchronous

   if (spres != ServerProcessing.Async)
   {
      if (lcc != null && lcc.GetData("priority") != null)
      {
         Console.WriteLine("  -> Post-execution change back to {0}",oldprio);
         Thread.CurrentThread.Priority = oldprio;
      }
   }
   return spres;
}
```

The sink provider for the server-side sink is quite straightforward. It looks more or less the same as those for the previous IServerChannelSinks.

On the client side, some minor inconveniences stem from this approach. Remember that you implemented an IMessageSink and not an IClientChannelSink in this case. Looking for an IMessageSinkProvider will not give you any results, so you'll have to implement an IClientChannelSink provider in this case as well—even though the sink is in reality an IMessageSink. The problem with this can be seen when looking at the following part of the IClientChannelSinkProvider interface:

```
IClientChannelSink CreateSink(IChannelSender channel,
     string url,
     object remoteChannelData);
```

This indicates `CreateSink()` has to return an IClientChannelSink in any case, even if your sink only needs to implement IMessageSink. You now have to extend your IMessageSink to implement IClientChannelSink as well. You also have to use caution because IClientChannelSink defines more methods that have to be implemented. Those methods are called when the sink is used as a channel sink (that is, after the formatter) and not as a message sink. You might not want to allow your users to position the sink *after* the formatter (because it wouldn't work there because it's changing the IMessage object's content), so you want to throw exceptions in those methods.

The complete client-side PriorityEmitterSink, which throws those exceptions when used in the wrong sequence, is shown in Listing 13-12.

Listing 13-12. *The Complete PriorityEmitterSink*

```
using System;
using System.Collections;
using System.IO;
using System.Runtime.Remoting;
using System.Runtime.Remoting.Channels;
using System.Runtime.Remoting.Messaging;
using System.Threading;

namespace PrioritySinks
{
 public class PriorityEmitterSink : BaseChannelObjectWithProperties,
                                    IClientChannelSink, IMessageSink
  {
   private IMessageSink _nextMsgSink;

   public IMessageCtrl AsyncProcessMessage(IMessage msg, IMessageSink replySink)
   {
      // only for method calls
      if (msg as IMethodCallMessage != null)
      {
         LogicalCallContext lcc =
            (LogicalCallContext) msg.Properties["__CallContext"];
         lcc.SetData("priority",Thread.CurrentThread.Priority);
         return _nextMsgSink.AsyncProcessMessage(msg,replySink);
      }
      else
      {
         return _nextMsgSink.AsyncProcessMessage(msg,replySink);
      }
   }
```

```
public IMessage SyncProcessMessage(IMessage msg)
{
   // only for method calls
   if (msg as IMethodCallMessage != null)
   {
      LogicalCallContext lcc =
         (LogicalCallContext) msg.Properties["__CallContext"];
      lcc.SetData("priority",Thread.CurrentThread.Priority);
      return _nextMsgSink.SyncProcessMessage(msg);
   }
   else
   {
      return _nextMsgSink.SyncProcessMessage(msg);
   }
}

public PriorityEmitterSink (object next)
{
   if (next as IMessageSink != null)
   {
      _nextMsgSink = (IMessageSink) next;
   }
}


public IMessageSink NextSink
{
   get
   {
      return _nextMsgSink;
   }
}

public IClientChannelSink NextChannelSink
{
   get
   {
      throw new RemotingException("Wrong sequence.");
   }
}

public void AsyncProcessRequest(IClientChannelSinkStack sinkStack,
   IMessage msg,
   ITransportHeaders headers,
   Stream stream)
{
   throw new RemotingException("Wrong sequence.");
}
```

```
   public void AsyncProcessResponse(
      IClientResponseChannelSinkStack sinkStack,
      object state,
      ITransportHeaders headers,
      Stream stream)
   {
      throw new RemotingException("Wrong sequence.");
   }

   public System.IO.Stream GetRequestStream(IMessage msg,
      ITransportHeaders headers)
   {
      throw new RemotingException("Wrong sequence.");
   }

   public void ProcessMessage(IMessage msg,
      ITransportHeaders requestHeaders,
      Stream requestStream,
      out ITransportHeaders responseHeaders,
      out Stream responseStream)
   {
      throw new RemotingException("Wrong sequence.");
   }
 }
}
```

The client-side PriorityEmitterSinkProvider, which is shown in Listing 13-13, is quite straightforward to implement. The only interesting method is `CreateSink()`.

Listing 13-13. *The Client-Side PriorityEmitterSinkProvider*

```
using System;
using System.Collections;
using System.Runtime.Remoting.Channels;

namespace PrioritySinks
{

   public class PriorityEmitterSinkProvider: IClientChannelSinkProvider
   {

      private IClientChannelSinkProvider next = null;

      public PriorityEmitterSinkProvider(IDictionary properties,
          ICollection providerData)
      {
         // not needed
      }
```

```
      public IClientChannelSink CreateSink(IChannelSender channel,
          string url, object remoteChannelData)
      {
         IClientChannelSink nextsink =
            next.CreateSink(channel,url,remoteChannelData);

         return new PriorityEmitterSink(nextsink);
      }

      public IClientChannelSinkProvider Next
      {
         get { return next; }
         set { next = value; }
      }

   }
}
```

Because the server-side sink shown in Listing 13-14 is an IServerChannelSink and not an IMessageSink, as is the client-side sink, the implementation is more consistent. You don't need to implement any additional interface here.

Listing 13-14. *The Server-Side PriorityChangerSink*

```
using System;
using System.Collections;
using System.IO;
using System.Runtime.Remoting;
using System.Runtime.Remoting.Messaging ;
using System.Runtime.Remoting.Channels;
using System.Threading;

namespace PrioritySinks
{

   public class PriorityChangerSink : BaseChannelObjectWithProperties,
      IServerChannelSink, IChannelSinkBase
   {

      private IServerChannelSink _next;

      public PriorityChangerSink (IServerChannelSink next)
      {
         _next = next;
      }

      public void AsyncProcessResponse (
                IServerResponseChannelSinkStack sinkStack,
                Object state,
```

```
            IMessage msg,
            ITransportHeaders headers,
            Stream stream)
{
   // restore the priority
   ThreadPriority priority = (ThreadPriority) state;
   Console.WriteLine("  -> Post-execution change back to {0}",priority);
   Thread.CurrentThread.Priority = priority;
}

public Stream GetResponseStream (IServerResponseChannelSinkStack sinkStack,
            Object state,
            IMessage msg,
            ITransportHeaders headers )
{
   return null;
}

public ServerProcessing ProcessMessage(IServerChannelSinkStack sinkStack,
   IMessage requestMsg,
   ITransportHeaders requestHeaders,
   Stream requestStream,
   out IMessage responseMsg,
   out ITransportHeaders responseHeaders,
   out Stream responseStream)
{
   LogicalCallContext lcc =
      (LogicalCallContext) requestMsg.Properties["__CallContext"];

   // storing the current priority
   ThreadPriority oldprio = Thread.CurrentThread.Priority;

   // check if the logical call context contains "priority"
   if (lcc != null && lcc.GetData("priority") != null)
   {
      // fetch the priority from the call context
      ThreadPriority priority =
         (ThreadPriority) lcc.GetData("priority");

      Console.WriteLine("-> Pre-execution priority change {0} to {1}",
         oldprio.ToString(),priority.ToString());

      // set the priority
      Thread.CurrentThread.Priority = priority;
   }
```

```
            // push on the stack and pass the call to the next sink
            // the old priority will be used as "state" for the response
            sinkStack.Push(this,oldprio);

            ServerProcessing spres =  _next.ProcessMessage (sinkStack,
               requestMsg, requestHeaders, requestStream,
               out responseMsg,out responseHeaders,out responseStream);

            // restore priority if call is not asynchronous

            if (spres != ServerProcessing.Async)
            {
               if (lcc != null && lcc.GetData("priority") != null)
               {
                  Console.WriteLine("-> Post-execution change back to {0}",oldprio);
                  Thread.CurrentThread.Priority = oldprio;
               }
            }
            return spres;
         }

         public IServerChannelSink NextChannelSink
         {
            get {return _next;}
            set {_next = value;}
         }
      }
   }
```

The corresponding server-side sink provider, which implements IServerChannelSinkProvider, is shown in Listing 13-15.

Listing 13-15. *The Server-Side PriorityChangerSinkProvider*

```
using System;
using System.Collections;
using System.Runtime.Remoting.Channels;

namespace PrioritySinks
{
   public class PriorityChangerSinkProvider: IServerChannelSinkProvider
   {
      private IServerChannelSinkProvider next = null;

      public PriorityChangerSinkProvider(IDictionary properties,
         ICollection providerData)
      {
         // not needed
      }
```

```
      public void GetChannelData (IChannelDataStore channelData)
      {
         // not needed
      }

      public IServerChannelSink CreateSink (IChannelReceiver channel)
      {
         IServerChannelSink nextSink = next.CreateSink(channel);
         return new PriorityChangerSink(nextSink);
      }

      public IServerChannelSinkProvider Next
      {
         get { return next; }
         set { next = value; }
      }

   }
}
```

To test this sink combination, use the following SAO, which returns the server-side thread's current priority:

```
public class TestSAO: MarshalByRefObject
{
   public String getPriority()
   {
      return System.Threading.Thread.CurrentThread.Priority.ToString();
   }
}
```

This SAO is called several times with different client-side thread priorities. The configuration file that is used by the server is shown here:

```
<configuration>
  <system.runtime.remoting>
    <application>
      <channels>
        <channel ref="http" port="5555">

          <serverProviders>
            <formatter ref="soap" />
                  <provider
             type="PrioritySinks.PriorityChangerSinkProvider, PrioritySinks" />
          </serverProviders>

        </channel>
      </channels>
```

```
      <service>
        <wellknown mode="Singleton"
               type="Server.TestSAO, Server" objectUri="TestSAO.soap" />
      </service>

    </application>
  </system.runtime.remoting>
</configuration>
```

The client-side configuration file will look like this:

```
<configuration>
 <system.runtime.remoting>
    <application>
       <channels>
        <channel ref="http">

          <clientProviders>
            <provider
               type="PrioritySinks.PriorityEmitterSinkProvider, PrioritySinks" />
            <formatter ref="soap" />
          </clientProviders>

        </channel>
     </channels>

     <client>
        <wellknown type="Server.TestSAO, generated_meta"
                   url="http://localhost:5555/TestSAO.soap" />
     </client>

   </application>
  </system.runtime.remoting>
</configuration>
```

For the test client, you can use SoapSuds to extract the metadata. When you run the application in Listing 13-16, you'll see the output shown in Figure 13-8.

Listing 13-16. *The Test Client*

```
using System;
using System.Runtime.Remoting;
using Server; // from generated_meta.dll
using System.Threading;

namespace Client
{
   delegate String getPrioAsync();

   class Client
   {
```

```
        static void Main(string[] args)
        {
            String filename = "client.exe.config";
            RemotingConfiguration.Configure(filename);

            TestSAO obj = new TestSAO();
            test(obj);

            Thread.CurrentThread.Priority = ThreadPriority.Highest;
            test(obj);

            Thread.CurrentThread.Priority = ThreadPriority.Lowest;
            test(obj);

            Thread.CurrentThread.Priority = ThreadPriority.Normal;
            test(obj);


            Console.ReadLine();
        }

        static void test(TestSAO obj)
        {
            Console.WriteLine("----------------- START TEST CASE ---------------");
            Console.WriteLine("  Local Priority: {0}",
                              Thread.CurrentThread.Priority.ToString());

            String priority1 = obj.getPriority();

            Console.WriteLine("   Remote priority: {0}",priority1.ToString());
            Console.WriteLine("----------------- END TEST CASE ---------------");
        }
    }
}
```

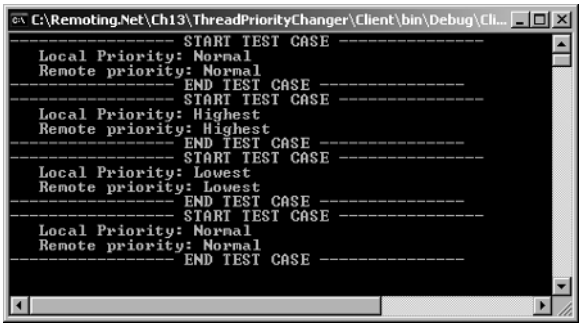

Figure 13-8. *The test client's output shows that the sinks work as expected.*

Changing the Programming Model

The previous sinks all add functionality to both the client- and the server-side of a .NET Remoting application. The pluggable sink architecture nevertheless also allows the creation of sinks, which change several aspects of the programming model. In Chapter 5, for example, you've seen that passing custom credentials such as username and password involves manual setting of the channel sink's properties for each object.

```
CustomerManager mgr = new CustomerManager();
IDictionary props = ChannelServices.GetChannelSinkProperties(mgr);
props["username"] = "dummyremotinguser";
props["password"] = "12345";
```

In most real-world applications, it is nevertheless preferable to set these properties on a per-host basis, or set them according to the base URL of the destination object. In a perfect world, this would be possible using either configuration files or code, as in the following example:

```
<configuration>
  <system.runtime.remoting>
    <application>
     <channels>
      <channel ref="http">
      <clientProviders>

         <formatter ref="soap" />
            <provider type="UrlAuthenticationSink.UrlAuthenticationSinkProvider,
                 UrlAuthenticationSink">

               <url
                     base="http://localhost"
                     username="DummyRemotingUser"
                     password="12345"
               />

               <url
                     base="http://www.somewhere.org"
                     username="MyUser"
                     password="12345"
               />
            </provider>
      </clientProviders>
      </channel>
     </channels>
    </application>
  </system.runtime.remoting>
</configuration>
```

When setting these properties in code, you can simply omit the <url> entries from the configuration file and instead use the following lines to achieve the same behavior:

```
UrlAuthenticator.AddAuthenticationEntry(
   "http://localhost",
   "dummyremotinguser",
   "12345");

UrlAuthenticator.AddAuthenticationEntry(
   "http://www.somewhere.org",
   "MyUser",
   "12345");
```

In fact, this behavior is not supported by default but can be easily implemented using a custom IClientChannelSink.

Before working on the sink itself, you have to write a helper class that provides static methods to store and retrieve authentication entries for given base URLs. All those entries will be stored in an ArrayList and can be retrieved by passing a URL to the `GetAuthenticationEntry()` method. In addition, default authentication information that will be returned if none of the specified base URLs matches the current object's URL can be set as well. This helper class is shown in Listing 13-17.

Listing 13-17. *The UrlAuthenticator Stores Usernames and Passwords*

```
using System;
using System.Collections;

namespace UrlAuthenticationSink
{

   internal class UrlAuthenticationEntry
   {
      internal String Username;
      internal String Password;
      internal String UrlBase;

      internal UrlAuthenticationEntry (String urlbase,
         String user,
         String password)
      {
         this.Username = user;
         this.Password = password;
         this.UrlBase = urlbase.ToUpper();
      }
   }

   public class UrlAuthenticator
   {
      private static ArrayList _entries = new ArrayList();
      private static UrlAuthenticationEntry _defaultAuthenticationEntry;
```

```
      public static void AddAuthenticationEntry(String urlBase,
         String userName,
         String password)
      {
         _entries.Add(new UrlAuthenticationEntry(
            urlBase,userName,password));
      }

      public static void SetDefaultAuthenticationEntry(String userName,
         String password)
      {
         _defaultAuthenticationEntry = new UrlAuthenticationEntry(
            null,userName,password);
      }

      internal static UrlAuthenticationEntry GetAuthenticationEntry(String url)
      {
         foreach (UrlAuthenticationEntry entr in _entries)
         {
            // check if a registered entry matches the url-parameter
            if (url.ToUpper().StartsWith(entr.UrlBase))
            {
               return entr;
            }
         }

         // if none matched, return the default entry (which can be null as well)
         return _defaultAuthenticationEntry;
      }
   }
}
```

The sink itself calls a method that checks if an authentication entry exists for the URL of the current message. It then walks the chain of sinks until reaching the final transport channel sink, on which is set the properties that contain the correct username and password. It finally sets a flag for this object's sink so that this logic will be applied only once per sink chain. The complete source for this sink can be found in Listing 13-18.

Listing 13-18. *The UrlAuthenticationSink*

```
using System;
using System.Runtime.Remoting.Channels;
using System.Runtime.Remoting.Messaging;
using System.IO;


namespace UrlAuthenticationSink
{
```

```
public class UrlAuthenticationSink: BaseChannelSinkWithProperties,
                            IClientChannelSink
{
   private IClientChannelSink _nextSink;
   private bool _authenticationParamsSet;

   public UrlAuthenticationSink(IClientChannelSink next)
   {
      _nextSink = next;
   }

   public IClientChannelSink NextChannelSink
   {
      get {
         return _nextSink;
      }
   }


   public void AsyncProcessRequest(IClientChannelSinkStack sinkStack,
      IMessage msg,
      ITransportHeaders headers,
      Stream stream)
   {
      SetSinkProperties(msg);
      // don't push on the sinkstack because this sink doesn't need
      // to handle any replies!
      _nextSink.AsyncProcessRequest(sinkStack,msg,headers,stream);
   }

   public void AsyncProcessResponse(
      IClientResponseChannelSinkStack sinkStack,
      object state,
      ITransportHeaders headers,
      Stream stream)
   {
      // not needed
   }


   public Stream GetRequestStream(IMessage msg,
                                  ITransportHeaders headers)
   {
      return _nextSink.GetRequestStream(msg, headers);
   }
```

```
      public void ProcessMessage(IMessage msg,
                                 ITransportHeaders requestHeaders,
                                 Stream requestStream,
                                 out ITransportHeaders responseHeaders,
                                 out Stream responseStream)
      {

         SetSinkProperties(msg);

         _nextSink.ProcessMessage(msg,requestHeaders,requestStream,
            out responseHeaders,out responseStream);
      }

      private void SetSinkProperties(IMessage msg)
      {
         if (! _authenticationParamsSet)
         {
            String url = (String) msg.Properties["__Uri"];

            UrlAuthenticationEntry entr =
               UrlAuthenticator.GetAuthorizationEntry(url);

            if (entr != null)
            {
               IClientChannelSink last = this;

               while (last.NextChannelSink != null)
               {
                  last = last.NextChannelSink;
               }

               // last now contains the transport channel sink

               last.Properties["username"] = entr.Username;
               last.Properties["password"] = entr.Password;
            }

            _authenticationParamsSet = true;
         }
      }
   }
}
```

The corresponding sink provider examines the <url> entry, which can be specified in the configuration file *below* the sink provider.

```
<provider type="UrlAuthenticationSink.UrlAuthenticationSinkProvider,
       UrlAuthenticationSink">
```

```
    <url
       base="http://localhost"
       username="DummyRemotingUser"
       password="12345"
     />

</provider>
```

The sink provider will receive those entries via the providerData collection, which contains objects of type SinkProviderData. Every instance of SinkProviderData has a reference to a properties dictionary that allows access to the attributes (base, username, and password) of the entry.

When the base URL is set in the configuration file, it simply calls `UrlAuthenticator.AddAuthenticationEntry()`. If no base URL has been specified, it sets this username/password as the default authentication entry. You can see the complete source code for this provider in Listing 13-19.

Listing 13-19. *The UrlAuthenticationSinkProvider*

```
using System;
using System.Runtime.Remoting.Channels;
using System.Collections;

namespace UrlAuthenticationSink
{
   public class UrlAuthenticationSinkProvider: IClientChannelSinkProvider
   {
      private IClientChannelSinkProvider _nextProvider;

      public UrlAuthenticationSinkProvider(IDictionary properties,
              ICollection providerData)
      {
         foreach (SinkProviderData obj in providerData)
         {
            if (obj.Name == "url")
            {
               if (obj.Properties["base"] != null)
               {
                  UrlAuthenticator.AddAuthenticationEntry(
                     (String) obj.Properties["base"],
                     (String) obj.Properties["username"],
                     (String) obj.Properties["password"]);
               }
               else
               {
                  UrlAuthenticator.SetDefaultAuthenticationEntry(
                     (String) obj.Properties["username"],
                     (String) obj.Properties["password"]);
```

```
            }
         }

      }
   }

   public IClientChannelSinkProvider Next
   {
      get {return _nextProvider; }
      set {_nextProvider = value;}
   }

   public IClientChannelSink CreateSink(IChannelSender channel,
          string url,
          object remoteChannelData)
   {
      // create other sinks in the chain
      IClientChannelSink next = _nextProvider.CreateSink(channel,
         url,
         remoteChannelData);

      // put our sink on top of the chain and return it
      return new UrlAuthenticationSink(next);
   }
 }
}
```

Using This Sink

When using this sink, you can simply add it to your client-side sink chain in the configuration file, as shown here:

```
<configuration>
  <system.runtime.remoting>
    <application>
     <channels>
      <channel ref="http">

        <clientProviders>
          <formatter ref="soap" />
          <provider type="UrlAuthenticationSink.UrlAuthenticationSinkProvider,
                UrlAuthenticationSink" />
        </clientProviders>

      </channel>
     </channels>
    </application>
  </system.runtime.remoting>
</configuration>
```

■ **Note** This sink is an IClientChannelSink, so you have to place it *after* the formatter.

To specify a username/password combination for a given base URL, you can now add this authentication information to the configuration file by using one or more <url> entries inside the <provider> section.

```
<clientProviders>
   <formatter ref="soap" />
   <provider type="UrlAuthenticationSink.UrlAuthenticationSinkProvider,
                  UrlAuthenticationSink">
        <url
               base="http://localhost"
               username="DummyRemotingUser"
               password="12345"
         />
      </provider>
</clientProviders>
```

If you don't want to hard code this information, you can ask the user of your client program for the username/password and employ the following code to register it with this sink:

```
UrlAuthenticator.AddAuthenticationEntry(<url>, <username>, <password>);
```

To achieve the same behavior as that of the <url> entry in the previous configuration snippet, you use the following command:

```
UrlAuthenticator.AddAuthenticationEntry(
   "http://localhost",
   "dummyremotinguser",
   "12345");
```

Avoiding the BinaryFormatter Version Mismatch

Custom .NET Remoting sinks can also be used as a workaround for certain glitches inside the framework. You can use them to change the way the .NET Remoting framework treats several exception conditions.[2]

As you've read in Chapter 10, there is a misleading exception that might be triggered from time to time when you use the HttpChannel with the BinaryFormatter while hosting your server-side components in IIS. In some cases, IIS sends back detailed information for some errors in HTML format. It also uses the content-type text/html.

The binary formatter, however, ignores this content-type header and interprets the data as a binary message and tries to deserialize it. This deserialization fails (as there is no binary message), and the client only receives an exception telling it that the deserialization failed. However, this final exception does not contain any additional information about the root cause of the problem.

2. The following was inspired by a blog post by my friend Richard Blewett at `http://staff.develop.com/richardb/weblog`.

To work around this issue, you can create a custom sink that will be used between the client-side binary formatter and the client-side transport channel. This sink can intercept the response message and check its content-type header. If the header is "application/octet-stream", then the message contains a real binary message and the sink will just forward it. Otherwise, the sink will read the complete error message in text format and forward a matching exception to the call chain.

To implement this, let me first show you the TcpTrace output for an incorrect message in Figure 13-9 and the (incorrect) resulting exception message in Figure 13-10.

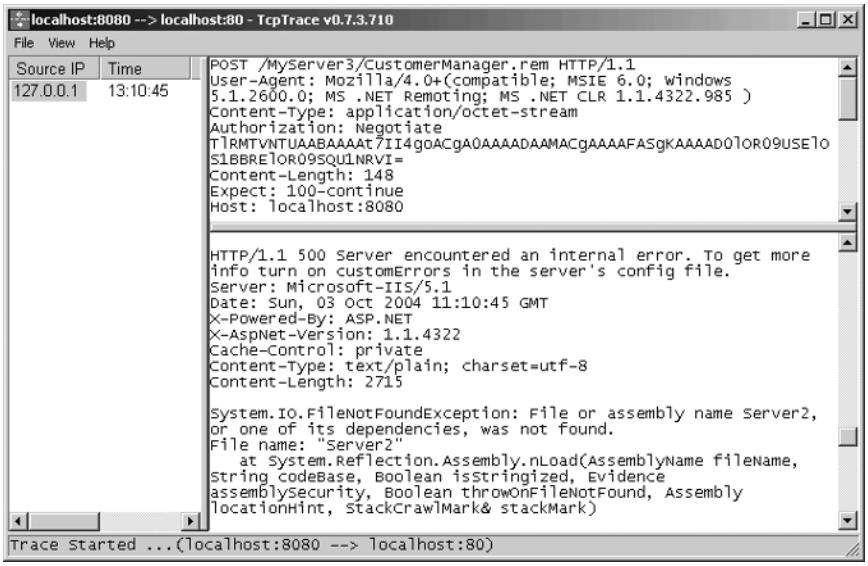
localhost:8080 --> localhost:80 - TcpTrace v0.7.3.710
File View Help
Source IP Time
127.0.0.1 13:10:45
POST /MyServer3/CustomerManager.rem HTTP/1.1
User-Agent: Mozilla/4.0+(compatible; MSIE 6.0; Windows 5.1.2600.0; MS .NET Remoting; MS .NET CLR 1.1.4322.985)
Content-Type: application/octet-stream
Authorization: Negotiate
TlRMTVNTUAABAAAAt7II4goACgA0AAAADAAMACgAAAAFASgKAAAAD0lOR09USElo
S1BBRElOR09SQU1NRVI=
Content-Length: 148
Expect: 100-continue
Host: localhost:8080
HTTP/1.1 500 Server encountered an internal error. To get more info turn on customErrors in the server's config file.
Server: Microsoft-IIS/5.1
Date: Sun, 03 Oct 2004 11:10:45 GMT
X-Powered-By: ASP.NET
X-AspNet-Version: 1.1.4322
Cache-Control: private
Content-Type: text/plain; charset=utf-8
Content-Length: 2715
System.IO.FileNotFoundException: File or assembly name Server2, or one of its dependencies, was not found.
File name: "Server2"
at System.Reflection.Assembly.nLoad(AssemblyName fileName, String codeBase, Boolean isStringized, Evidence assemblySecurity, Boolean throwOnFileNotFound, Assembly locationHint, StackCrawlMark& stackMark)
Trace Started ...(localhost:8080 --> localhost:80)

Figure 13-9. *This response message results in an incorrect exception.*

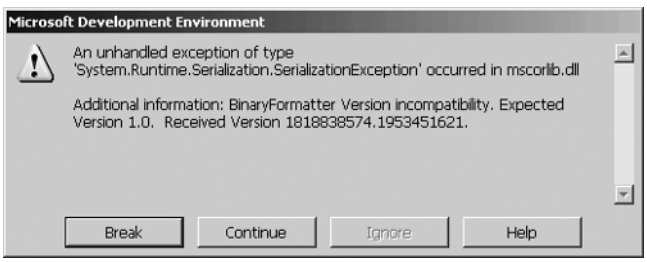

Figure 13-10. *The incorrect exception information*

The custom sink I am going to show you will intercept the preceding message and turn it into a more reasonable exception.

This sink's process message method will first forward the call onto the next sink in the chain (so that it is transferred to the server) and will then inspect the response stream. To inspect the stream, it uses a method, `GetExceptionIfNecessary()`, which I'll show you in just in a minute.

This method returns either null, if everything is OK, or an exception with a more meaningful error message.

If an exception is returned, `ProcessMessage()` will simply `throw` the exception.

```
public void ProcessMessage(IMessage msg,
   ITransportHeaders requestHeaders,
   Stream requestStream,
   out ITransportHeaders responseHeaders,
   out Stream responseStream)
{
   _next.ProcessMessage(msg, requestHeaders, requestStream,
                        out responseHeaders, out responseStream);
   Exception ex =
         GetExceptionIfNecessary(ref responseHeaders, ref responseStream);

   if (ex!=null) throw ex;
}
```

The method for examining the content of the response stream first looks at the "Content-Type" header. If this header has the value "application/octet-stream", it is not modified and the method does not return an exception. Otherwise, it reads the complete response text and creates a new Exception object, setting its description to the text that has been received from the server.

This method can look like this:

```
private Exception GetExceptionIfNecessary(
   ref ITransportHeaders headers, ref Stream stream)
{
   int chunksize=0x400;
   MemoryStream ms = new MemoryStream();

   string ct = headers["Content-Type"] as String;

   if (ct==null || ct != "application/octet-stream")
   {
      byte[] buf = new byte[chunksize];
      StringBuilder bld = new StringBuilder();
      for (int size = stream.Read(buf, 0, chunksize);
           size > 0; size = stream.Read(buf, 0, chunksize))
      {
         bld.Append(Encoding.ASCII.GetString(buf, 0, size));
      }
      return new RemotingException(bld.ToString());
   }
   return null;
}
```

Additionally, you will need to implement `AsyncProcessRequest()` and `AsyncProcessResponse()` to provide a similar behavior.

```
public void AsyncProcessRequest(IClientChannelSinkStack sinkStack,
   IMessage msg,
```

```
   ITransportHeaders headers,
   Stream stream)
{
   sinkStack.Push(this,null);
   _next.AsyncProcessRequest( sinkStack, msg, headers, stream);
}

public void AsyncProcessResponse(
   IClientResponseChannelSinkStack sinkStack,
   object state,
   ITransportHeaders headers,
   Stream stream)
{
   Exception ex = GetExceptionIfNecessary(ref headers, ref stream);
   if (ex!=null)
   {
      sinkStack.DispatchException(ex);
   }
   else
   {
      sinkStack.AsyncProcessResponse(headers, stream);
   }
}
```

After creating the complete sink and an appropriate sink provider, you can use it in your client-side configuration file like this:

```
<configuration>
   <system.runtime.remoting>
      <application>
         <channels>
            <channel ref="http">
               <clientProviders>
                  <formatter ref="binary" />
                  <provider
     type="HttpErrorInterceptor.InterceptorSinkProvider, HttpErrorInterceptor" />
               </clientProviders>
            </channel>
         </channels>
         <!-- client entries removed -->
      </application>
   </system.runtime.remoting>
</configuration>
```

■ **Note** You can find the complete source code for this sink (and, of course, for any other sample contained in this book) at the book's source-code download page at `http://www.apress.com`.

After you apply this configuration file and run the same client-side command that initially resulted in the incorrect exception, you will now receive the information shown in Figure 13-11. This information correctly reflects the source of the problem.

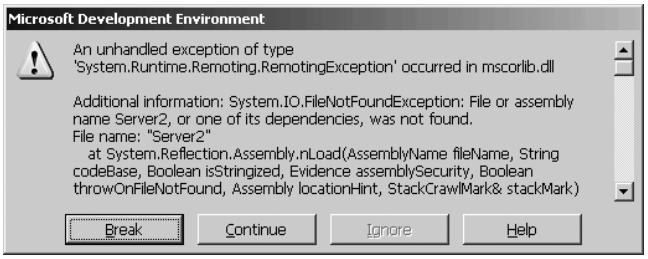
Microsoft Development Environment
An unhandled exception of type 'System.Runtime.Remoting.RemotingException' occurred in mscorlib.dll
Additional information: System.IO.FileNotFoundException: File or assembly name Server2, or one of its dependencies, was not found.
File name: "Server2"
at System.Reflection.Assembly.nLoad(AssemblyName fileName, String codeBase, Boolean isStringized, Evidence assemblySecurity, Boolean throwOnFileNotFound, Assembly locationHint, StackCrawlMark& stackMark)
Break
Continue
Ignore
Help

Figure 13-11. *The corrected exception information*

Using a Custom Proxy

In the previous parts of this chapter, you read about the possible ways you can extend the .NET Remoting framework using additional custom message sinks. There is another option for changing the default behavior of the remoting system: custom proxy objects. Figure 13-12 shows you the default proxy configuration.

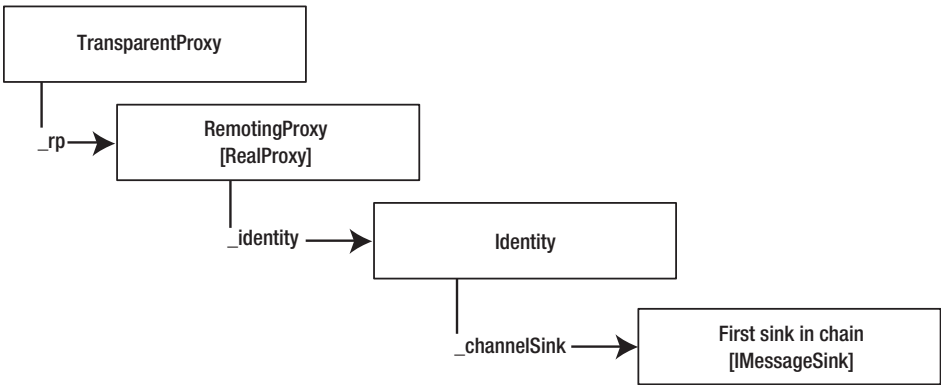

Figure 13-12. *The default combination of proxy objects*

You can change this by replacing RemotingProxy with a custom proxy that inherits from RealProxy.

■ **Note** You'll normally miss the opportunity to use configuration files in this case. To work around this issue, you can use the RemotingHelper class, discussed in Chapter 6.

To do this, you basically have to implement a class that extends RealProxy, provides a custom constructor, and overrides the `Invoke()` method to pass the message on to the correct message sink.

As shown in Listing 13-20, the constructor first has to call the base object's constructor and then checks all registered channels to determine whether they accept the given URL by calling their `CreateMessageSink()` methods. If a channel can service the URL, it returns an IMessageSink object that is the first sink in the remoting chain. Otherwise it returns `null`. The constructor will throw an exception if no registered channel is able to parse the URL.

Listing 13-20. *A Skeleton Custom Remoting Proxy*

```
using System;
using System.Collections;
using System.Runtime.Remoting;
using System.Runtime.Remoting.Channels;
using System.Runtime.Remoting.Channels.Http;
using System.Runtime.Remoting.Proxies;
using System.Runtime.Remoting.Messaging;

namespace Client
{
   public class CustomProxy: RealProxy
   {
      String _url;
      String _uri;
      IMessageSink _sinkChain;

      public CustomProxy(Type type, String url) : base(type)
      {
         _url = url;

         // check each registered channel if it accepts the
         // given URL
         IChannel[] registeredChannels = ChannelServices.RegisteredChannels;
         foreach (IChannel channel in registeredChannels )
         {
            if (channel is IChannelSender)
            {
               IChannelSender channelSender = (IChannelSender)channel;

               // try to create the sink
               _sinkChain = channelSender.CreateMessageSink(_url,
                  null, out _uri);

               // if the channel returned a sink chain, exit the loop
               if (_sinkChain != null) break;
            }
         }
```

```
            // no registered channel accepted the URL
            if (_sinkChain == null)
            {
               throw new Exception("No channel has been found for " + _url);
            }
         }

         public override IMessage Invoke(IMessage msg)
         {
            msg.Properties["__Uri"] = _url;

            // TODO: process the request message

            IMessage retMsg = _sinkChain.SyncProcessMessage(msg);

            // TODO: process the return message

            return retMsg;
         }
      }
}
```

To employ this proxy, you have to instantiate it using the special constructor by passing the Type of the remote object and its URL. You can then call `GetTransparentProxy()` on the resulting CustomProxy object and cast the returned TransparentProxy to the remote object's type, as shown in Listing 13-21.

Listing 13-21. *Using a Custom Proxy*

```
using System;
using System.Runtime.Remoting;
using System.Runtime.Remoting.Channels;
using System.Runtime.Remoting.Channels.Http;
using Service; // from service.dll

namespace Client
{
   class Client
   {
      static void Main(string[] args)
      {
         ChannelServices.RegisterChannel(new HttpChannel());

         CustomProxy prx = new CustomProxy(typeof(Service.SomeSAO),
            "http://localhost:1234/SomeSAO.soap");

         SomeSAO obj = (SomeSAO) prx.GetTransparentProxy();

         String res = obj.doSomething();
```

```
            Console.WriteLine("Got result: {0}",res);
            Console.ReadLine();
        }
    }
}
```

To show you an example for a custom proxy, I implement some methods that dump the request and return messages' contents. These methods are called from the proxy's `Invoke()`, which will be executed whenever your client calls a method on the TransparentProxy object. This is shown in Listing 13-22.

■ **Note** You can also call these content dump methods in an IMessageSink!

Listing 13-22. *Custom Proxy That Dumps the Request and Response Messages' Contents*

```
using System;
using System.Collections;
using System.Runtime.Remoting;
using System.Runtime.Remoting.Channels;
using System.Runtime.Remoting.Channels.Http;
using System.Runtime.Remoting.Proxies;
using System.Runtime.Remoting.Messaging;

namespace Client
{
    public class CustomProxy: RealProxy
    {
        String _url;
        String _uri;
        IMessageSink _sinkChain;

        public CustomProxy(Type type, String url) : base(type)
        {
            _url = url;

            // check each registered channel if it accepts the
            // given URL
            IChannel[] registeredChannels = ChannelServices.RegisteredChannels;
            foreach (IChannel channel in registeredChannels )
            {
                if (channel is IChannelSender)
                {
                    IChannelSender channelSender = (IChannelSender)channel;

                    // try to create the sink
                    _sinkChain = channelSender.CreateMessageSink(_url,
                        null, out _uri);
```

```
            // if the channel returned a sink chain, exit the loop
            if (_sinkChain != null) break;
         }
      }

      // no registered channel accepted the URL
      if (_sinkChain == null)
      {
         throw new Exception("No channel has been found for " + _url);
      }
   }

   public override IMessage Invoke(IMessage msg)
   {
      msg.Properties["__Uri"] = _url;
      DumpMessageContents(msg);
      IMessage retMsg = _sinkChain.SyncProcessMessage(msg);
      DumpMessageContents(retMsg);
      return retMsg;
   }

   private String GetPaddedString(String str)
   {
      String ret = str + "                    ";
      return ret.Substring(0,17);
   }

   private void DumpMessageContents(IMessage msg)
   {
      Console.WriteLine("=======================================");
      Console.WriteLine("============ Message Dump ==============");
      Console.WriteLine("=======================================");

      Console.WriteLine("Type: {0}", msg.GetType().ToString());

      Console.WriteLine("--- Properties ---");
      IDictionary dict = msg.Properties;
      IDictionaryEnumerator enm =
             (IDictionaryEnumerator) dict.GetEnumerator();

      while (enm.MoveNext())
      {
         Object key = enm.Key;
         String keyName = key.ToString();
         Object val = enm.Value;

         Console.WriteLine("{0}: {1}", GetPaddedString(keyName), val);
```

```
            // check if it's an object array
            Object[] objval = val as Object[];
            if (objval != null)
            {
               DumpObjectArray(objval);
            }

         }

         Console.WriteLine();
         Console.WriteLine();
      }

      private void DumpObjectArray(object[] data)
      {
         // if empty -> return
         if (data.Length == 0) return;

         Console.WriteLine("\t --- Array Contents ---");
         for (int i = 0; i < data.Length; i++)
         {
            Console.WriteLine("\t{0}: {1}", i, data[i]);
         }
      }
   }
}
```

The output of the preceding client when used with this proxy is shown in Figure 13-13.

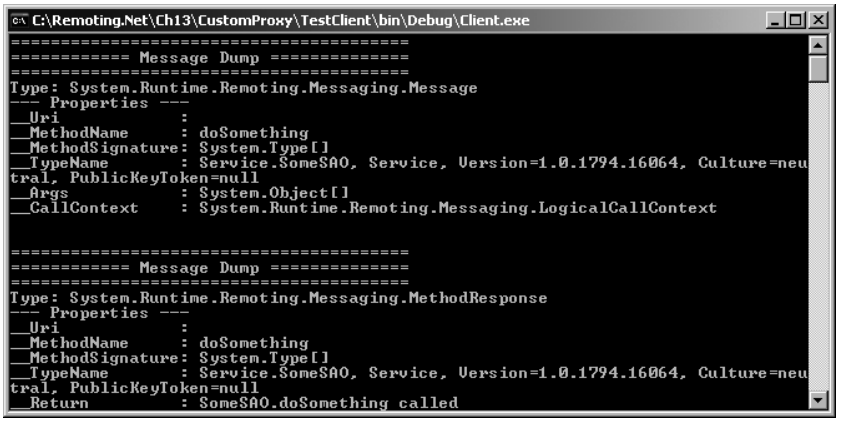

Figure 13-13. *Using the custom proxy to dump the messages' contents*

As nearly all the functionality you'll want to implement in a common .NET Remoting scenario can be implemented with IMessageSinks, IClientChannelSinks, or IServerChannelSinks, I suggest you implement functionality by using these instead of custom proxies in .NET Remoting. Sinks provide the additional benefits of being capable of working with configuration files and being chainable. This chaining allows you to develop a set of very focused sinks that can then be combined to solve your application's needs.

Custom proxies are nevertheless interesting because they can also be used for local objects. In this case, you don't have to implement a special constructor, only override `Invoke()`. You can then pass any MarshalByRefObject to another constructor (which is provided by the parent RealProxy) during creation of the proxy. All method calls to this local object then pass the proxy as an IMessage object and can therefore be processed. You can read more on message-based processing for local applications in Chapter 11.

Some Final Words of Caution

Custom .NET Remoting sinks should enhance the transport protocol, but should not normally provide application-specific functionality. Or to rephrase it: you should not implement business logic in custom sinks. The reason is that this would tie your business logic code to the transport protocol you are using.

Experience shows that business applications outlive their initial environments. In my consulting practice, I have seen several applications developed with VB4 for example, then ported to VB5, then to VB6. Subsequently, some parts have been ported to VB .NET, but some parts have simply been wrapped as COM DLLs to expose their functionality to .NET and Web Services applications. These pieces of code have definitely outlived their initial environments.

If you therefore tie your business logic code to some side effects of your custom sinks, you'll make it harder for your code to live without the .NET Remoting framework. However, this might become critical as in the future you might want to expose the same components via Web Services, or by using the upcoming stack of Indigo technologies.

All of these alternative protocols also have means of extensibility. This means you would not just have to port your business logic (which will usually be a fairly trivial task), but also your custom sinks. The latter might prove harder, as the extensibility models of these other technologies are completely different.

So, to summarize: custom .NET Remoting sinks are a great means to enhance the distributed application protocol used in your application. However, they should in most cases *not* be used to create side effects that unnecessarily tie your business logic code to the .NET Remoting framework. If you decide that it makes sense for your project to do this nevertheless, you have to take into account that migrating your code might be more difficult in the future.

Summary

In this chapter you have seen how you can leverage the .NET Remoting framework's extensibility. You should now be able to apply the internals shown in Chapters 11 and 12 to extend and customize the .NET Remoting programming model to suit your needs.

You now know the differences between IMessageSink, which is used before the message reaches the client-side formatter, and IClientChannelSink, which is used after the serialization of the IMessage object. You know that you can add properties to the IMessage object's

LogicalCallContext to pass it to the server, where it can be read by an IServerChannelSink, and you can also encrypt or compress a request by using a combination of IClientChannelSinks and IServerChannelSinks.

You also learned how sink providers are developed, and that a client-side IMessageSink has to be created by an IClientChannelSinkProvider as well and therefore has to implement the IClientChannelSink's methods. Finally, you read about custom proxies, which allow you to implement additional functionality before the message reaches the chain of sinks.

In the next chapter, you get a chance to use the knowledge gained here to implement a complete transport channel from scratch.

CHAPTER 14

Developing a Transport Channel

In the last three chapters you've seen how the .NET Remoting framework's various layers can be extended. The ultimate example of customization is nevertheless the creation of a specific transport channel.

As you already know by now, the .NET Remoting framework comes out of the box with three transport channels: HTTP channel, IPC channel, and TCP channel. Even though these protocols cover a lot of common scenarios, you might have a special need for other transport channels for some applications. On one hand, protocols like named pipes,[1] for example, could allow for higher performance (with security!) for applications communicating with each other on the same machine. On the other hand, you might have the need for asynchronous communication with MSMQ or even by standard Internet e-mail using SMTP and POP3.

No matter which protocol you choose, you must watch for one thing: most protocols are *either* synchronous *or* asynchronous in their nature. As the remoting framework supports both kinds of calls, you'll need to either map asynchronous .NET Remoting calls onto a synchronous protocol like HTTP or TCP or map synchronous .NET Remoting calls onto an asynchronous protocol like MSMQ or SMTP/POP3.

With the former, you will, for example, have to spawn a secondary thread that waits for the reply and notifies the IAsyncResult (as seen in Chapter 7) whenever a response is returned. The latter demands that you block the calling thread and wake it up as soon as the asynchronous response has been received from the underlying protocol.

Protocol Considerations

First, I have to place a disclaimer here: this chapter will provide you with the necessary ideas and walk you through the process of designing and developing a specialized transport channel. It isn't the objective of this chapter to give you a commercial grade SMTP/POP3 channel. Even though you *could* just use this channel in your application, you'll need to understand it fully before doing so because neither the author nor the publisher will provide any support when you include this channel in your applications.

1. Which will be provided by the IpcChannel, which will be shipped with version 2.0 of the .NET Framework.

Okay, now let's start with the implementation of this channel. Before doing this you nevertheless have to *know* the protocol (or at least the relevant parts of it) that you are going to use. Every Internet protocol has its so-called request-for-comment (RFC) documents. The SMTP protocol in its most recent version is shown in RFC2821 and POP3 in RFC1939. You can get them at `http://www.ietf.org/rfc/rfc2821.txt` and `http://www.ietf.org/rfc/rfc1939.txt`.

You can generally search for any RFC at `http://www.faqs.org/rfcs/index.html`, but you should keep in mind that normally RFCs are identified by the full protocol name and not the more common acronym. For example, you'd have to search for "Simple Mail Transfer Protocol", not "SMTP", to find the mentioned RFCs. If you don't know the real protocol name, you can search for the abbreviation on `http://www.webopedia.com/`.

Generally the transfer of e-mails is split between two protocols: SMTP and POP3. SMTP is used to send e-mails from your mail client to a mail server and will then be used for inter–mail-server communication until it reaches a destination mailbox. POP3 is used to receive e-mails from a server's mailbox to your e-mail client.

The Shortcut Route to SMTP . . .

To save you from having to read through the whole RFC2821, I provide here a short summary of the relevant parts needed to implement this channel. First, SMTP uses a request/reply syntax normally sent over a TCP connection to port 25 on the server. The client can send a list of commands, and the server replies to them with a status code and a message. The status code is a three-digit number of which the first digit specifies the class. These are shown in Table 14-1. The message that might be sent after the status code is not standardized and should be ignored or used only for reporting errors back to the user.

Table 14-1. *SMTP Response Code Classes*

Response Code	Meaning
2xx	Positive response. Command accepted and executed.
3xx	Intermediate or transient positive. Command accepted, but more information is needed. In plain English, this means the client can now start to transfer the mail (if received as a reply to the DATA command).
4xx	Transient error. Try again later.
5xx	Permanent error. You quite certainly did something wrong!

A successful SMTP conversion can therefore look like the following code. (The ➤ symbol indicates that the client sends this line and the ◄ symbol indicates the server's reply.) You can easily try this out yourself by entering **telnet <mailserver> 25**—but be aware that the commands you input might not be echoed back to you.

```
◄ 220 MyRemotingServer Mercury/32 v3.21c ESMTP server ready.
➤ HELO localhost
◄ 250 MyRemotingServer Hi there, localhost.
➤ MAIL FROM: <client_1@localhost>
◄ 250 Sender OK - send RCPTs.
➤ RCPT TO: <server_1@localhost>
◄ 250 Recipient OK - send RCPT or DATA.
➤ DATA
```

```
◄ 354 OK, send data, end with CRLF.CRLF
► <sending message contents inclusive headers>
►_ sending <CR><LF>.<CR><LF> (i.e. a "dot" between two CR/LFs)
◄ 250 Data received OK.
► QUIT
◄ 221 MyRemotingServer Service closing channel.
```

As you can see here, several commands can be sent by the client. At the beginning of the session, after the server announces itself with `220 servername message`, the client will send `HELO hostname`. This starts the SMTP session, and the server responds with `250 servername message`.

For each e-mail the client wants to send via this server, the following process takes place. First, the client starts with `MAIL FROM: <e-mail address>` (note that the e-mail address *has* to be enclosed in angle brackets). The server replies with `250 message` if it allows e-mails from this sender; otherwise, it replies with a 4xx or 5xx status code. The client then sends one or more `RCPT TO: <e-mail address>` commands that designate the recipients of this e-mail and that are also confirmed by `250 message` replies.

As soon as all recipients have been specified, the client sends `DATA` to notify the server that it's going to send the e-mail's content. The server replies with `354 message` and expects the client to send the e-mail and finish with "." (dot) on a single new line (that is, the client sends <CR><LF><DOT><CR><LF>). The server then acknowledges the e-mail by replying with `250 message`.

At this point the client can send further e-mails by issuing `MAIL FROM` or can terminate the session with the `QUIT` command, which will be confirmed by the server via a `221 message` reply. The server will then also close the TCP connection. Sometime after the message is sent, it will be placed in a user's mailbox from where it can be retrieved by the POP3 protocol.

. . . And Round-Trip to POP3

Generally POP3 works in quite the same way as SMTP: it's also a request/response protocol. POP3 messages are generally sent over a TCP connection to port 110. Instead of the status codes SMTP relies on, POP3 supports three kinds of replies. The first two are `+OK message` to indicate success and `-ERR message` to indicate failure. The messages after the status code aren't standardized, and as such should be used only to report errors to your user and not be parsed by a program.

Another type of reply is a *content reply*, which is used whenever you request information from the server that might span multiple lines. In this case the server will indicate the end of the response with the same <CR><LF><DOT><CR><LF> sequence that is used by SMTP to end the transfer of the e-mail text.

A sample POP3 session might look like this. To start it, enter **telnet <mailserver> 110**.

```
◄ +OK <1702038211.21388@vienna01>, MercuryP/32 v3.21c ready.
►_ USER server_1
◄ +OK server_1 is known here.
►_ PASS server_1
◄ +OK Welcome! 1 messages (231 bytes)
►_ LIST
◄ +OK 1 messages, 3456 bytes
◄ 1 231
```

```
◄ .
➤_ RETR 1
◄ +OK Here it comes...
◄ <e-mail text>
◄ .
➤_ DELE 1
◄ +OK Message deleted.
➤_ QUIT
◄ +OK vienna01 Server closing down.
```

As you can see in this connection trace, the client authenticates itself at the server by sending the commands `USER username` and `PASS password`. In this case the password is transported in clear text over the network. Most POP3 servers also support an APOP command based on an MD5 digest that incorporates a server-side timestamp to disable replay attacks. You can read more about this in RFC1939.

The server then replies with a message like `+OK Welcome! 1 messages (231 bytes)`. You should never try to parse this reply to receive the message count; instead, either send the command `STAT`, which will be answered by `+OK number_of_messages total_bytes` or issue a `LIST` command, which will first return `+OK message` and then return this line once for each message: `message_number bytes`. The reply concludes with a <CR><LF><DOT><CR><LF> sequence.

The client can then issue a `RETR message_number` command to retrieve the content of a specific message (with a final <CR><LF><DOT><CR><LF> as well) and a `DELE message_number` statement to delete it at the server. This deletion is only carried out after sending `QUIT` to the server, which then closes the connection.

Character Encoding Essentials

After reading the last few pages, you are almost fully equipped to start with the design of your channel. The last thing you need to know before you can start to write some code is how the resulting e-mail has to look.

Because there's no standard for the binding of .NET Remoting to custom transfer protocols, I just elected Simon Fell's recommendation for SOAP binding to SMTP as the target specification for this implementation. You can find the latest version of this document at `http://www.pocketsoap.com/specs/smtpbinding/`. In essence, it basically says that the content has to be either Base64 or Quoted-Printable encoded and needs to supply a given number of e-mail headers. So, what does this mean? Essentially, the e-mail format we know today was designed ages ago when memory was expensive, WAN links were slow, and computers liked to deal with 7-bit ASCII characters.

Nowadays we instead use Unicode, which allows us to have *huge* numbers of characters so that even languages like Japanese, Chinese, or Korean can be encoded. This is, of course, far from 7 bit, so you have to find a way to bring such data back to a 7-bit format of "printable characters." Base64 does this for you; it is described in detail in section 5.2 of RFC1521, available at `http://www.ietf.org/rfc/rfc1521.txt`. To encode a given byte array with the Base64 algorithm, you can use `Convert.ToBase64String()`.

Creating E-Mail Headers

An e-mail contains not only the body, but also header information. For a sample SOAP-via-SMTP request, the complete e-mail might look like this:

```
From: client_1@localhost
To: server_1@localhost
Message-Id: <26fc4f4cd8de4567a66ccea6897dc481@REMOTING>
MIME-Version: 1.0
Content-Type: text/xml; charset=utf-8
Content-Transfer-Encoding: BASE64

...encoded SOAP request here...
```

To match a response message to the correct request, the value of the Message-Id header will be included in the In-Reply-To header of the response message.

```
From: server_1@localhost
To: client_1@localhost
Message-Id: <97809278530983552398576545869067@REMOTING>
In-Reply-To: <26fc4f4cd8de4567a66ccea6897dc481@REMOTING>
MIME-Version: 1.0
Content-Type: text/xml; charset=utf-8
Content-Transfer-Encoding: BASE64

...encoded SOAP response here...
```

You also need to include some special headers that are taken from the ITransportHeaders object of the .NET Remoting request. Those will be preceded by X-REMOTING- so that a complete remoting request e-mail might look like this:

```
From: client_1@localhost
To: server_1@localhost
Message-Id: <26fc4f4cd8de4567a66ccea6897dc481@REMOTING>
MIME-Version: 1.0
Content-Type: text/xml; charset=utf-8
Content-Transfer-Encoding: BASE64
X-REMOTING-Content-Type: text/xml; charset="utf-8"
X-REMOTING-SOAPAction:
 "http://schemas.microsoft.com/clr/ns/System.Runtime    ➥
.Remoting.Activation.IActivator#Activate"
X-REMOTING-URI: /RemoteActivationService.rem

...encoded .NET Remoting request here...
```

The encoded .NET Remoting request itself can be based on either the binary formatter or SOAP!

Encapsulating the Protocols

Now that you've got some of the protocol basics down, you're ready for the source code for this channel. Okay, here goes. First you have to encapsulate the SMTP and POP3 protocols that are later used in the client-side and server-side transport channel sinks.

The first part of this is shown in Listing 14-1. This source file encapsulates the lower-level SMTP protocol. It provides a public constructor that needs the hostname of your SMTP-server as a parameter. Its `SendMessage()` method takes the sender's and recipient's e-mail address and the full e-mail's text (including the headers) as parameters. It then connects to the SMTP server and sends the specified mail.

Listing 14-1. *Encapsulating the Lower-Level SMTP Protocol*

```
using System;
using System.Net.Sockets;
using System.Net;
using System.IO;
using System.Text;

namespace SmtpChannel
{
   public class SmtpConnection
   {
      private String _hostname;
      private TcpClient _SmtpConnection;
      private NetworkStream _smtpStream;
      private StreamReader _smtpResponse;

      public SmtpConnection(String hostname) {
         _hostname = hostname;
      }

      private void Connect() {
         _SmtpConnection = new TcpClient(_hostname,25);
         _smtpStream = _SmtpConnection.GetStream();
         _smtpResponse = new StreamReader(_smtpStream);
      }

      private void Disconnect() {
         _smtpStream.Close();
         _smtpResponse.Close();
         _SmtpConnection.Close();
      }

      private void SendCommand(String command, int expectedResponseClass) {
         // command: the SMTP command to send
         // expectedResponseClass: first digit of the expected smtp response
```

```
            // throws an exception if the server's responsecode's first
            // digit (resonse class) is > the expectedResponseClass.

            // if expectedResponseClass == 0, it will be ignored

            command = command + "\r\n";
            byte[] cmd = Encoding.ASCII.GetBytes(command);
            _smtpStream.Write(cmd,0,cmd.Length);
            String response = _smtpResponse.ReadLine();

            if (expectedResponseClass != 0) {
               int resp = Convert.ToInt32(response.Substring(0,1));
               if (resp > expectedResponseClass) {
                  throw new Exception("SMTP Server returned unexpected " +
                     "response:\n'" + response + "'");
               }
            }
         }

         public void SendMessage(String from, String to, String text) {
            try
            {
               Connect();
               SendCommand("HELO localhost",2);
               SendCommand("MAIL FROM: <" + from + ">",2);
               SendCommand("RCPT TO: <" + to + ">",2);
               SendCommand("DATA",3);
               byte[] bodybytes = Encoding.ASCII.GetBytes(text + "\r\n");
               _smtpStream.Write(bodybytes,0,bodybytes.Length);
               SendCommand(".",3);
               SendCommand("QUIT",0);
            } finally {
               try {
                  Disconnect();
               } catch (Exception e) {/*ignore*/};
            }
         }

      }
   }
```

To encapsulate the POP3 protocol, you first have to add a class that will hold the parsed message and its headers. This is shown in Listing 14-2.

Listing 14-2. *A Retrieved and Parsed Message*

```
using System;
using System.Collections;
```

```
namespace SmtpChannel
{
   public class POP3Msg
   {
      public String From;
      public String To;
      public String Body;
      public String Headers;
      public String MessageId;
      public String InReplyTo;
   }
}
```

The helper class POP3Connection encapsulates the lower-level details of the POP3 protocol. After construction, it connects to the server, authenticates the user, and issues a `LIST` command to retrieve the list of messages waiting.

```
using System;
using System.Net.Sockets;
using System.Net;
using System.IO;
using System.Collections;
using System.Text;

namespace SmtpChannel
{
   public class POP3Connection
   {
      private class MessageIndex
      {
         // will be used to store the result of the LIST command
         internal int Number;
         internal int Bytes;

         internal MessageIndex(int num, int msgbytes)
         {
            Number = num;
            Bytes  = msgbytes;
         }
      }

      private String _hostname;
      private String _username;
      private String _password;

      private TcpClient _pop3Connection;
      private NetworkStream _pop3Stream;
      private StreamReader _pop3Response;
      private IDictionary _msgs;
```

```
public POP3Connection(String hostname, String username, String password)
{
   // try to connect to the server with the supplied username
   // and password.

   _hostname = hostname;
   _username = username;
   _password = password;
   try
   {
      Connect();
   }
   catch (Exception e)
   {
      try
      {
         Disconnect();
      }
      catch (Exception ex) {/* ignore */}

      throw e;
   }
}

private void Connect()
{
   // initialize the list of messages
   _msgs = new Hashtable();

   // open the connection
   _pop3Connection = new TcpClient(_hostname,110);
   _pop3Stream = _pop3Connection.GetStream();
   _pop3Response = new StreamReader(_pop3Stream);

   // ignore first line (server's greeting)
   String response = _pop3Response.ReadLine();

   // authenticate
   SendCommand("USER " + _username,true);
   SendCommand("PASS " + _password,true);

   // retrieve the list of messages
   SendCommand("LIST",true);
   response = _pop3Response.ReadLine();
   while (response != ".")
   {
      // add entries to _msgs dictionary
      int pos = response.IndexOf(" ");
```

```
            String msgnumStr = response.Substring(0,pos);
            String bytesStr = response.Substring(pos);

            int msgnum = Convert.ToInt32(msgnumStr);
            int bytes = Convert.ToInt32(bytesStr);

            MessageIndex msgidx = new MessageIndex(msgnum,bytes);
            _msgs.Add (msgidx,msgnum);
            response = _pop3Response.ReadLine();

        }
    }
```

These methods make use of the `SendCommand()` method, which sends a specified POP3 command to the server and checks the response if indicated.

```
private void SendCommand(String command,bool needOK)
{
    // sends a single command.

    // if needOK is set it will check the response to begin
    // with "+OK" and will throw an exception if it doesn't.

    command = command + "\r\n";
    byte[] cmd = Encoding.ASCII.GetBytes(command);

    // send the command
    _pop3Stream.Write(cmd,0,cmd.Length);
    String response = _pop3Response.ReadLine();

    // check the response
    if (needOK)
    {
       if (!response.Substring(0,3).ToUpper().Equals("+OK"))
       {
          throw new Exception("POP3 Server returned unexpected " +
            "response:\n'" + response + "'");
       }
    }
}
```

The `MessageCount` property returns the number of messages available at the server.

```
public int MessageCount
{
   get
```

```
   {
      // returns the message count after connecting and
      // issuing the LIST command
      return _msgs.Count;
   }
}
```

GetMessage() returns a POP3Msg object, which is filled by retrieving the specified message from the server. It does this by sending the RETR command to the POP3 server and by checking for special e-mail headers. It then populates the POP3Msg object's properties with those headers and the e-mail body.

```
public POP3Msg GetMessage(int msgnum)
{
   // create the resulting object
   POP3Msg tmpmsg = new POP3Msg();

   // retrieve a single message
   SendCommand("RETR " + msgnum,true);
   String response = _pop3Response.ReadLine();

   // read the response line by line and populate the
   // correct properties of the POP3Msg object

   StringBuilder headers = new StringBuilder();
   StringBuilder body = new StringBuilder();
   bool headersDone=false;
   while ((response!= null) && (response != "." ))
   {
      // check if all headers have been read
      if (!headersDone)
      {
         if (response.Length >0)
         {
            // this will only parse the headers which are relevant
            // for .NET Remoting

            if (response.ToUpper().StartsWith("IN-REPLY-TO:"))
            {
               tmpmsg.InReplyTo = response.Substring(12).Trim();
            }
            else if (response.ToUpper().StartsWith("MESSAGE-ID:"))
            {
               tmpmsg.MessageId = response.Substring(11).Trim();
            }
            else if (response.ToUpper().StartsWith("FROM:"))
            {
               tmpmsg.From = response.Substring(5).Trim();
            }
```

```
            else if (response.ToUpper().StartsWith("TO:"))
            {
               tmpmsg.To = response.Substring(3).Trim();
            }
            headers.Append(response).Append("\n");
         }
         else
         {
            headersDone = true;
         }
      }
      else
      {
         // all headers have been read, add the rest to
         // the body.

         // For .NET Remoting, we need the body in a single
         // line to decode Base64, therefore no <CR><LF>s will
         // be appended!

         body.Append(response);
      }

      // read next line
      response = _pop3Response.ReadLine();
   }

   // set the complete header and body Strings of POP3Msg
   tmpmsg.Body = body.ToString();
   tmpmsg.Headers = headers.ToString();
   return tmpmsg;
}
```

What's still needed is `DeleteMessage()`, which flags a message for deletion by sending the `DELE` command to the server.

```
public void DeleteMessage(int msgnum)
{
   // issue the DELE command to delete the specified message
   SendCommand("DELE " + msgnum,true);
}
```

And finally, you need a method to send the `QUIT` command and disconnect from the server. This `Disconnect()` method is shown here:

```
public void Disconnect()
{
   // sends QUIT and disconnects
```

```
   try
   {
      // send QUIT to commit the DELEs
      SendCommand("QUIT",false);
   }
   finally
   {
      // close the connection
      _pop3Stream.Close();
      _pop3Response.Close();
      _pop3Connection.Close();
   }
}
```

With those two helper classes presented previously, you'll be able to access SMTP and POP3 servers without having to deal further with the details of those protocols. Nevertheless, what's still missing is a mapping between the .NET Remoting framework and those e-mail messages.

Checking for New Mail

As POP3 will not notify the client whenever a message has been received, it is necessary to continuously log on to the server to check for new e-mails. This is done by the POP3Polling class. Each instance of this class contacts a given server in regular intervals. If a message is available, it will fetch it and invoke a delegate on another helper class that will then process this message.

The POP3Polling class can be run in two modes: client or server mode. Whenever it's in client mode, it only starts polling after the client has sent a request to the server. As soon as the server's reply is handled, this class stops polling to save network bandwidth. In server mode, it checks the server regularly to handle any remoting requests.

As you can see in the following part of this class, it provides a custom constructor that accepts the server's hostname, username, password, the polling interval, and a flag indicating whether it should run in server or client mode:

```
using System;
using System.Collections;
using System.Threading;

namespace SmtpChannel
{
   public class POP3Polling
   {
      delegate void HandleMessageDelegate(POP3Msg msg);

      // if it's a remoting server, we will poll forever
      internal bool _isServer;
```

```
// if this is not a server, this property has to be set to
// start polling
internal bool _needsPolling;

// is currently polling
internal bool _isPolling;

// polling interval in seconds
internal int _pollInterval;

// logon data
private String _hostname;
private String _username;
private String _password;

internal POP3Polling(String hostname, String username, String password,
        int pollInterval, bool isServer)
{
   _hostname = hostname;
   _username = username;
   _password = password;
   _pollInterval = pollInterval;
   _isServer = isServer;

   if (!_isServer) { _needsPolling = false; }
}
```

The following `Poll()` method does the real work. It is started in a background thread and checks for new e-mails as long as either `_isServer` or `_needsPolling` (which indicates the need for client-side polling) is true. While it polls the server, it also sets a flag to prevent reentrancy.

`SMTPHelper.MessageReceived()` is a static method that is called by using an asynchronous delegate in a *fire-and-forget* way. This method will handle the e-mail and forward it to the remoting framework. Have a look at it here:

```
private void Poll()
{
   if (_isPolling) return;
   _isPolling = true;
   do
   {
      Thread.Sleep(_pollInterval * 1000);

      POP3Connection pop = new POP3Connection(_hostname,_username,_password);
      for (int i =1;i<=pop.MessageCount;i++)
      {
         POP3Msg msg =  pop.GetMessage(i);
         HandleMessageDelegate del = new HandleMessageDelegate(
               SMTPHelper.MessageReceived);
```

```
            del.BeginInvoke(msg,null,null);
            pop.DeleteMessage(i);
         }
         pop.Disconnect();
         pop = null;
      } while (_isServer || _needsPolling);
      _isPolling = false;
   }
```

The last method of this class, `CheckAndStartPolling()`, can be called externally to start the background thread as soon as `_needsPolling` has been changed. It does the necessary checks to ensure that only one background thread is running per instance.

```
internal void CheckAndStartPolling()
{
   if (_isPolling) return;

   if (_isServer || _needsPolling)
   {
      Thread thr = new Thread(new ThreadStart(this.Poll));
      thr.Start();
      thr.IsBackground = true;
   }
}
```

Registering a POP3 Server

When creating a custom channel like this, you might want to be able to have one application that can be client and server at the same time. To optimize the polling strategy in this case, you want to have only one instance of the POP3Polling class for a specified e-mail address, and not two separate ones for the client channel and the server channel.

For client-only use of this channel, you also want to have a central point that can be notified as soon as a request has been sent to the server so it can start the background thread to check for new e-mails. Additionally, this class needs a counter of responses that have been sent and for which no reply has yet been received. As soon as all open replies have been handled, it should again notify the POP3Polling object to stop checking for those e-mails.

This class provides a method, `RegisterPolling()`, that has to be called for each registered channel to create the POP3Polling object. This method first checks whether the same username/hostname combination has already been registered. If this is not the case, it creates the new object and stores a reference to it in a Hashtable.

If the same combination has been registered before, this method first checks whether the new registration request is for a server-side channel and in this case switches the already known POP3Polling object into server mode. It also checks the polling intervals of the old and the new object and takes the lower value. Finally, it calls the POP3Polling object's `CheckAndStartPolling()` method to enable the background thread if it is in server mode.

```
using System;
using System.Collections;
using System.Threading;
```

```
namespace SmtpChannel
{
   public class POP3PollManager
   {
      // dictionary of polling instances
      static IDictionary _listeners = Hashtable.Synchronized(new Hashtable());

      // number of sent messages for which no response has been received
      private static int _pendingResponses;
      private static int _lastPendingResponses;

      public static void RegisterPolling(String hostname, String username,
         String password, int pollInterval, bool isServer)
      {
         String key = username + "|" + hostname;

         // check if this combination has already been registered
         POP3Polling pop3 = (POP3Polling) _listeners[key];
         if (pop3 == null)
         {
            // create a new listener
            pop3 = new POP3Polling(hostname,username,password,
               pollInterval,isServer);

            _listeners[key]= pop3;
         }
         else
         {
            // change to server-mode if needed
            if (!pop3._isServer && isServer)
            {
               pop3._isServer = true;
            }

            // check for pollInterval => lowest interval will be taken
            if (! (pop3._pollInterval > pollInterval))
            {
               pop3._pollInterval = pollInterval;
            }
         }
         pop3.CheckAndStartPolling();
      }
```

Two other methods that are provided in this class have to be called to notify all registered POP3Polling objects that the remoting framework is waiting for a reply or that it has received a reply.

```
internal static void RequestSent()
{
   _pendingResponses++;
   if (_lastPendingResponses<=0 && _pendingResponses > 0)
   {
      IEnumerator enmr = _listeners.GetEnumerator();
      while (enmr.MoveNext())
      {
         DictionaryEntry entr = (DictionaryEntry) enmr.Current;
         POP3Polling pop3 = (POP3Polling) entr.Value;
         pop3._needsPolling = true;
         pop3.CheckAndStartPolling();
      }
   }
   _lastPendingResponses = _pendingResponses;
}

internal static void ResponseReceived()
{
   _pendingResponses--;
   if (_pendingResponses <=0)
   {
      IEnumerator enmr = _listeners.GetEnumerator();
      while (enmr.MoveNext())
      {
         DictionaryEntry entr = (DictionaryEntry) enmr.Current;
         POP3Polling pop3 = (POP3Polling) entr.Value;
         pop3._needsPolling = false;
      }
   }
   _lastPendingResponses = _pendingResponses;
}
```

Connecting to .NET Remoting

What you've seen until now has been quite protocol specific, because I haven't yet covered any connections between the underlying protocol to .NET Remoting. This task is handled by the SMTPHelper class. This class holds three synchronized Hashtables containing the following data:

- Objects that are waiting for a response to a given SMTP message ID. These can be either Thread objects or SmtpChannel.AsyncResponseHandler objects, both of which are shown later. These are stored in `_waitingFor`.
- The server-side transport sink for any e-mail address that has been registered with a SMTPServerChannel in `_servers`.
- The received responses that will be cached while waking up the thread that has been blocked is stored in `_responses`. This is a short-term storage that is only used for the fractions of a second it takes for the thread to wake up and fetch and remove the response.

This class also starts with a method that transforms the .NET Remoting–specific information in the form of a stream and an ITransportHeaders object into an e-mail message. This method writes the standard e-mail headers, adds the remoting-specific headers from the ITransportHeaders object, encodes the stream's contents to Base64, and ensures a maximum line length of 73 characters. Finally, it sends the e-mail using the SmtpConnection helper class.

```
using System;
using System.Text;
using System.IO;
using System.Collections;
using System.Runtime.Remoting.Channels;
using System.Threading;

namespace SmtpChannel
{

   public class SMTPHelper
   {

      // threads waiting for response
      private static IDictionary _waitingFor =
         Hashtable.Synchronized(new Hashtable());

      // known servers
      private static IDictionary _servers =
         Hashtable.Synchronized(new Hashtable());

      // responses received
      private static IDictionary _responses =
         Hashtable.Synchronized(new Hashtable());


      // sending messages
      private static void SendMessage(String ID,String replyToId,
         String mailfrom, String mailto, String smtpServer,
         ITransportHeaders headers, Stream stream, String objectURI)
      {
         StringBuilder msg = new StringBuilder();

         if (ID != null)
         {
            msg.Append("Message-Id: ").Append(ID).Append("\r\n");
         }
         if (replyToId != null)
         {
            msg.Append("In-Reply-To: ").Append(replyToId).Append("\r\n");
         }
```

```
msg.Append("From: ").Append(mailfrom).Append("\r\n");
msg.Append("To: ").Append(mailto).Append("\r\n");
msg.Append("MIME-Version: 1.0").Append("\r\n");
msg.Append("Content-Type: text/xml; charset=utf-8").Append("\r\n");
msg.Append("Content-Transfer-Encoding: BASE64").Append("\r\n");

// write the remoting headers
IEnumerator headerenum = headers.GetEnumerator();
while (headerenum.MoveNext())
{
   DictionaryEntry entry = (DictionaryEntry) headerenum.Current;
   String key = entry.Key as String;
   if (key == null || key.StartsWith("__"))
   {
      continue;
   }
   msg.Append("X-REMOTING-").Append(key).Append(": ");
   msg.Append(entry.Value.ToString()).Append("\r\n");
}

if (objectURI != null)
{
   msg.Append("X-REMOTING-URI: ").Append(objectURI).Append("\r\n");
}

msg.Append("\r\n");
MemoryStream fs = new MemoryStream();

byte[] buf = new Byte[1000];
int cnt = stream.Read(buf,0,1000);
int bytecount = 0;
while (cnt>0)
{
   fs.Write(buf,0,cnt);
   bytecount+=cnt;
   cnt = stream.Read(buf,0,1000);
}


// convert the whole string to Base64 encoding
String body = Convert.ToBase64String(fs.GetBuffer(),0,bytecount);

// and ensure the maximum line length of 73 characters
int linesNeeded = (int) Math.Ceiling(body.Length / 73);
```

```
            for (int i = 0;i<=linesNeeded;i++)
            {
               if (i != linesNeeded)
               {
                  String line = body.Substring(i*73,73);
                  msg.Append(line).Append("\r\n");
               }
               else
               {
                  String line = body.Substring(i*73);
                  msg.Append(line).Append("\r\n");
               }
            }

            // send the resulting message
            SmtpConnection con = new SmtpConnection (smtpServer);
            con.SendMessage(mailfrom,mailto,msg.ToString());
         }
```

This method is not called directly by the framework, but instead the class provides two other methods that are made for this purpose. The first one, named `SendRequestMessage()`, generates a message ID that is later returned using an out parameter and then calls `SendMessage()` to send the e-mail via SMTP. It next calls the POP3PollManager's `RequestSent()` method so that the background thread will start checking for incoming reply mails.

The second method, `SendReplyMessage()`, takes a given message ID and sends a reply:

```
internal static void SendRequestMessage(String mailfrom, String mailto,
   String smtpServer, ITransportHeaders headers, Stream request,
   String objectURI, out String ID)
{
   ID = "<" + Guid.NewGuid().ToString().Replace("-","") + "@REMOTING>";
   SendMessage(ID,null,mailfrom,mailto,smtpServer,headers,request,objectURI);
   POP3PollManager.RequestSent();
}

internal static void SendResponseMessage(String mailfrom, String mailto,
   String smtpServer, ITransportHeaders headers, Stream response,
   String ID)
{
   SendMessage(null,ID,mailfrom,mailto,smtpServer,headers,response,null);
}
```

The more complex part of mapping the underlying protocol to the .NET Remoting framework is the handling of responses. As the combination of SMTP and POP3 is asynchronous in its nature, whereas most .NET calls are executed synchronously, you have to provide a means for blocking the underlying thread until the matching response has been received. This is accomplished by the following method, which adds the waiting thread to the `_waitingFor` Hashtable and suspends it afterwards. Whenever a response is received (which is handled by another function running in a different thread), the response is stored in the `_responses` Hashtable and the thread awakened again. It then fetches the value from `_responses` and returns it to the caller.

```
internal static POP3Msg WaitAndGetResponseMessage(String ID)
{
   // suspend the thread until the message returns
   _waitingFor[ID] = Thread.CurrentThread;

   Thread.CurrentThread.Suspend();

   // waiting for resume
   POP3Msg pop3msg = (POP3Msg) _responses[ID];
   _responses.Remove(ID);
   return pop3msg;
}
```

The previous method showed you the handling of synchronous .NET Remoting messages, but you might also want to support asynchronous delegates. In this case, a callback object is placed in `_waitingFor` and the method that handles incoming POP3 messages simply invokes the corresponding method on this callback object.

```
internal static void RegisterAsyncResponseHandler(String ID,
   AsyncResponseHandler ar)
{
   _waitingFor[ID] = ar;
}
```

One of the most important methods is `MessageReceived()`, which is called by POP3Polling as soon as an incoming message has been collected from the server. This method attempts to map all incoming e-mails to .NET Remoting calls.

There are two quite different types in e-mails: requests that are sent from a client to a server and reply messages that are sent back from the server. The distinction between these is that a reply e-mail includes an In-Reply-To header, whereas the request message only contains a Message-Id header.

If the incoming message is a request message, the `MessageReceived()` method checks the `_servers` Hashtable to retrieve the server-side sink chain for the destination e-mail address. It will then call `HandleIncomingMessage()` on this SMTPServerTransportSink object.

For reply messages, on the other hand, the method checks whether any objects are waiting for an answer to the contained In-Reply-To header. If the waiting object is a thread, the POP3Msg object will be stored in the `_responses` Hashtable and the thread will be awakened. For asynchronous calls, the waiting object will be of type AsyncResponseHandler. In this case the framework will simply call its `HandleAsyncResponsePop3Msg()` method. Regardless of whether the reply has been received for a synchronous or an asynchronous message, `POP3PollManager.ResponseReceived()` will be called to decrement the count of unanswered messages and to eventually stop polling if all replies have been received.

```
internal static void MessageReceived(POP3Msg pop3msg)
{
   // whenever a pop3 message has been received, it
   // will be forwarded to this method

   // check if it's a request or a reply
   if ((pop3msg.InReplyTo == null) && (pop3msg.MessageId != null))
```

```
{
   // it's a request

   String requestID = pop3msg.MessageId;

   // Request received

   // check for a registered server
   SMTPServerTransportSink snk = (SMTPServerTransportSink)
      _servers[GetCleanAddress(pop3msg.To)];

   if (snk==null)
   {
      // No server side sink found for address
      return;
   }

   // Dispatch the message to serversink
   snk.HandleIncomingMessage(pop3msg);

}
else if (pop3msg.InReplyTo != null)
{
   // a response must contain the in-reply-to header
   String responseID = pop3msg.InReplyTo.Trim();

   // check who's waiting for it
   Object notify = _waitingFor[responseID];

   if (notify as Thread != null)
   {
      _responses[responseID] = pop3msg;

      // Waiting thread found. Will wake it up
      _waitingFor.Remove(responseID );
      ((Thread) notify).Resume();
      POP3PollManager.ResponseReceived();
   }
   else if (notify as AsyncResponseHandler != null)
   {
      _waitingFor.Remove(responseID);
      POP3PollManager.ResponseReceived();
      ((AsyncResponseHandler)notify).HandleAsyncResponsePop3Msg(
         pop3msg);
   }
   else
```

```
      {
         // no one is waiting for this reply. ignore.
      }
   }
}
```

Another method is employed to map a POP3Msg object onto the objects that .NET Remoting needs: Stream and ITransportHeader. `ProcessMessage()` does this by taking all custom remoting headers starting with X-REMOTING- from the e-mail and putting them into a new ITransportHeader object. It then converts back the Base64-encoded payload into a byte array and creates a new MemoryStream that allows the remoting framework to read the byte array. It also returns the message ID as an out parameter.

```
internal static void ProcessMessage(POP3Msg pop3msg,
   out ITransportHeaders headers, out Stream stream, out String ID)
{
   // this method will split it into a TransportHeaders and
   // a Stream object and will return the "remoting ID"

   headers = new TransportHeaders();


   // first all remoting headers (which start with "X-REMOTING-")
   // will be extracted and stored in the TransportHeaders object
   String tmp = pop3msg.Headers;
   int pos = tmp.IndexOf("\nX-REMOTING-");
   while (pos >= 0)
   {
      int pos2 = tmp.IndexOf("\n",pos+1);
      String oneline = tmp.Substring(pos+1,pos2-pos-1);

      int poscolon = oneline.IndexOf(":");
      String key = oneline.Substring(11,poscolon-11).Trim();
      String headervalue = oneline.Substring(poscolon+1).Trim();
      if (key.ToUpper() != "URI")
      {
         headers[key] = headervalue;
      }
      else
      {
         headers["__RequestUri"] = headervalue;
      }
      pos = tmp.IndexOf("\nX-REMOTING-",pos2);
   }

   String fulltext = pop3msg.Body ;
   fulltext = fulltext.Trim();
   byte[] buffer = Convert.FromBase64String(fulltext);
```

```
   stream=new MemoryStream(buffer);

   ID = pop3msg.MessageId;
}
```

There's also a method called `RegisterServer()` in SMTPHelper that stores a server-side sink chain in the `_servers` Hashtable using the e-mail address as a key.

```
public static void RegisterServer(SMTPServerTransportSink snk,
   String address)
{
   // registering sink for a specified e-mail address
   _servers[address] = snk;
}
```

The last two methods in the helper class are of a more generic nature. The first parses a URL in the form smtp:someone@somedomain.com/URL/to/object and returns the e-mail address separated from the object's URI (which is /URL/to/object in this case) as out parameters.

```
internal static void parseURL(String url, out String email,
   out String objectURI)
{
   // format:   "smtp:user@host.domain/URL/to/object"

   // is split to:
   //      email = user@host.domain
   //      objectURI = /URL/to/object
   int pos = url.IndexOf("/");
   if (pos > 0)
   {
      email = url.Substring(5,pos-5);
      objectURI = url.Substring(pos);
   }
   else if (pos ==-1)
   {
      email = url.Substring(5);
      objectURI ="";
   }
   else
   {
      email = null;
      objectURI = url;
   }

}
```

The second method is used to parse an e-mail address for an incoming request. It accepts addresses in a variety of formats and returns the generic form of user@domain.com:

```
public static String GetCleanAddress(String address)
{
   // changes any kind of address like "someone@host"
   // "<someone@host>" "<someone@host> someone@host"
   // to a generic format of "someone@host"

   address = address.Trim();
   int posAt = address.IndexOf("@");
   int posSpaceAfter = address.IndexOf(" ",posAt);
   if (posSpaceAfter != -1) address = address.Substring(0,posSpaceAfter);

   int posSpaceBefore = address.LastIndexOf(" ");

   if (posSpaceBefore != -1 && posSpaceBefore < posAt)
   {
      address = address.Substring(posSpaceBefore+1);
   }

   int posLt = address.IndexOf("<");
   if (posLt != -1)
   {
      address = address.Substring(posLt+1);
   }

   int posGt = address.IndexOf(">");
   if (posGt != -1)
   {
      address = address.Substring(0,posGt);
   }

   return address;
}
```

Implementing the Client Channel

Before looking into the implementation of the client-side SMTPClientChannel, let me show you how it will later be used in a configuration file.

```
<channel
      name="smtpclient"
      type="SmtpChannel.SMTPClientChannel, SmtpChannel"
      senderEmail="client_1@localhost"
      smtpServer="localhost"
      pop3Server="localhost"
      pop3User="client_1"
      pop3Password="client_1"
      pop3PollInterval="1"
>
```

All of these parameters are mandatory, and I explain their meanings in Table 14-2.

Table 14-2. *Parameters for SMTPClientChannel*

Parameter	Description
name	Unique name for this channel.
senderEmail	The value for the e-mail's From: header. The server will reply to the address specified here.
smtpServer	Your outgoing e-mail server's name.
pop3Server	Your incoming mail server's name.
pop3User	The POP3 user account that is assigned to this client application.
pop3Password	The password for the application's POP3 account.
pop3PollInterval	Interval in seconds at which the framework will check for new mail at the server.

A client channel has to extend BaseChannelWithProperties and implement IChannelSender, which in turn extends IChannel. These interfaces are shown in Listing 14-3.

Listing 14-3. *IChannel and IChannelSender*

```
public interface IChannel
{
    // properties
    string ChannelName { get; }
    int ChannelPriority { get; }

    // methods
    string Parse(string url, ref String objectURI);
}

public interface IChannelSender: IChannel
{

    // methods
    IMessageSink CreateMessageSink(string url, object remoteChannelData,
       ref String objectURI);
}
```

Let's have a look at the basic implementation of a channel and the IChannel interface first. Each channel that is creatable using a configuration file has to supply a special constructor that takes two parameters: an IDictionary object, which will contain the attributes specified in the configuration file (for example, smtpServer); and an IClientChannelSinkProvider object, which points to the first entry of the chain of sink providers specified in the configuration file.

In the case of the SMPTClientChannel, this constructor will store those values to member variables and call `POP3PollManager.RegisterPolling()` to register the connection information.

```
using System;
using System.Collections;
```

```
using System.Runtime.Remoting.Channels;
using System.Runtime.Remoting;
using System.Runtime.Remoting.Messaging;

namespace SmtpChannel
{
   public class SMTPClientChannel: BaseChannelWithProperties, IChannelSender
   {
      IDictionary _properties;
      IClientChannelSinkProvider _provider;
      String _name;

      public SMTPClientChannel (IDictionary properties,
                  IClientChannelSinkProvider clientSinkProvider)
      {
         _properties = properties;
         _provider = clientSinkProvider;
         _name = (String) _properties["name"];

         POP3PollManager.RegisterPolling(
            (String) _properties["pop3Server"],
            (String) _properties["pop3User"],
            (String) _properties["pop3Password"],
            Convert.ToInt32((String)_properties["pop3PollInterval"]),
            false);
      }
```

The implementation of IChannel itself is quite straightforward. You basically have to return a priority and a name for the channel, both of which can be either configurable or hard coded. You also have to implement a `Parse()` method that takes a URL as its input parameter. It then has to check if the given URL is valid for this channel (returning null if it isn't) and split it into its base URL, which is the return value from the method, and the object's URI, which is returned as an out parameter.

```
public string ChannelName
{
   get
   {
      return _name;
   }
}

public int ChannelPriority
{
   get
   {
      return 0;
   }
}
```

```
public string Parse(string url, out string objectURI)
{
   String email;
   SMTPHelper.parseURL(url, out email, out objectURI);
   if (email == null || email=="" || objectURI == null || objectURI =="")
   {
      return null;
   }
   else
   {
      return "smtp:" + email;
   }
}
```

The implementation of IChannelSender consists only of a single method: `CreateMessageSink()`. This method will *either* receive a URL *or* a channel data store as parameters and will return an IMessageSink as a result and the destination object's URI as an out parameter.

When no URL is specified as a parameter, you should cast the channel data store (which is passed as object) to IChannelDataStore and take the first URL from it instead. You then have to check whether the URL is valid for your channel and return null if it isn't. Next you add the client channel's transport sink provider at the end of the provider chain and call `CreateSink()` on the first provider. The resulting sink chain is then returned from the method.

```
public IMessageSink CreateMessageSink(string url, object remoteChannelData,
      out string objectURI)
{

   if (url == null && remoteChannelData != null &&
         remoteChannelData as IChannelDataStore != null )
   {
      IChannelDataStore ds = (IChannelDataStore) remoteChannelData;
      url = ds.ChannelUris[0];
   }

   // format:   "smtp:user@host.domain/URI/to/object"
   if (url != null && url.ToLower().StartsWith("smtp:"))
   {
      // walk to last provider and this channel sink's provider
      IClientChannelSinkProvider prov = _provider;
      while (prov.Next != null) { prov = prov.Next ;};

      prov.Next = new SMTPClientTransportSinkProvider(
           (String) _properties["senderEmail"],
           (String) _properties["smtpServer"]);

      String dummy;
      SMTPHelper.parseURL(url,out dummy,out objectURI);
```

```
            IMessageSink msgsink =
                    (IMessageSink) _provider.CreateSink(this,url,remoteChannelData);

            return msgsink;
        }
        else
        {
            objectURI =null;
            return null;
        }
    }
```

Creating the Client's Sink and Provider

Even though this sink provider is called a transport sink provider, it is in fact a straightforward implementation of an IClientChannelSinkProvider, which you've encountered in Chapter 13. The main difference is that its `CreateSink()` method has to parse the URL to provide the transport sink with the correct information regarding the destination e-mail address and the object's URI. It also doesn't need to specify any special constructors, as it will not be initialized from a configuration file. The complete sink provider is shown in Listing 14-4.

Listing 14-4. *The SMTPClientTransportSinkProvider*

```
using System;
using System.Collections;
using System.Runtime.Remoting.Channels;
using System.Runtime.Remoting;
using System.Runtime.Remoting.Messaging;

namespace SmtpChannel
{
    public class SMTPClientTransportSinkProvider: IClientChannelSinkProvider
    {

        String _senderEmailAddress;
        String _smtpServer;

        public SMTPClientTransportSinkProvider(String senderEmailAddress,
                String smtpServer)
        {
            _senderEmailAddress = senderEmailAddress;
            _smtpServer = smtpServer;
        }

        public IClientChannelSink CreateSink(IChannelSender channel,
            string url, object remoteChannelData)
```

```
      {
         String destinationEmailAddress;
         String objectURI;
         SMTPHelper.parseURL(url,out destinationEmailAddress,out objectURI);

         return new SMTPClientTransportSink(destinationEmailAddress,
              _senderEmailAddress,_smtpServer, objectURI);
      }

      public IClientChannelSinkProvider Next
      {
         get
         {
            return null;
         }
         set
         {
            // ignore as this has to be the last provider in the chain
         }
      }
   }
}
```

When relying on the helper classes presented previously, the client-side sink's implementation will be quite simple as well. First, you have to provide a constructor that allows the sender's and the recipient's e-mail address, the object's URI, and the SMTP server to be set.

```
using System;
using System.Runtime.Remoting.Channels;
using System.Runtime.Remoting;
using System.Runtime.Remoting.Messaging;
using System.IO;

namespace SmtpChannel
{

   public class SMTPClientTransportSink: BaseChannelSinkWithProperties,
      IClientChannelSink, IChannelSinkBase
   {
      String _destinationEmailAddress;
      String _senderEmailAddress;
      String _objectURI;
      String _smtpServer;

      public SMTPClientTransportSink(String destinationEmailAddress,
         String senderEmailAddress, String smtpServer, String objectURI)
      {
         _destinationEmailAddress = destinationEmailAddress;
         _senderEmailAddress = senderEmailAddress;
```

```
        _objectURI = objectURI;
        _smtpServer = smtpServer;
    }
```

The key functionality of this sink is that ProcessMessage() and AsyncProcessMessage() cannot just forward the parameters to another sink, but instead have to send it by e-mail to another process. ProcessMessage() parses the URL to split it into the e-mail address and the object's URI. Those values are then used to call SMTPHelper.SendRequestMessage(). As this method has to block until a response is received, it also calls SMTPHelper.WaitAndGetResponseMessage(). Finally, it hands over the processing of the return message to the ProcessMessage() method of SMTPHelper to split it into a stream and an ITransportHeaders object that have to be returned from ProcessMessage() as an out parameter.

```
public void ProcessMessage(IMessage msg,
   ITransportHeaders requestHeaders, Stream requestStream,
   out ITransportHeaders responseHeaders,
   out Stream responseStream)
{
   String ID;
   String objectURI;
   String email;

   // check the URL
   String URL = (String) msg.Properties["__Uri"];
   SMTPHelper.parseURL(URL,out email,out objectURI);

   if ((email==null) || (email == ""))
   {
      email = _destinationEmailAddress;
   }

   // send the message
   SMTPHelper.SendRequestMessage(_senderEmailAddress,email,_smtpServer,
           requestHeaders,requestStream,objectURI, out ID);

   // wait for the response
   POP3Msg popmsg = SMTPHelper.WaitAndGetResponseMessage(ID);

   // process the response
   SMTPHelper.ProcessMessage(popmsg,out responseHeaders,out responseStream,
        out ID);
}
```

The AsyncProcessMessage() method does not block and wait for a reply, but instead creates a new instance of AsyncResponseHandler (which I introduce in a bit) and passes the sink stack to it. It then registers this object with the SMTPHelper to enable it to forward the response to the underlying sinks and finally to the IAsyncResult object that has been returned from the delegate's BeginInvoke() method.

```
public void AsyncProcessRequest(IClientChannelSinkStack sinkStack,
   IMessage msg, ITransportHeaders headers, Stream stream)
{
   String ID;
   String objectURI;
   String email;

   // parse the url
   String URL = (String) msg.Properties["__Uri"];
   SMTPHelper.parseURL(URL,out email,out objectURI);

   if ((email==null) || (email == ""))
   {
      email = _destinationEmailAddress;
   }

   // send the request message
   SMTPHelper.SendRequestMessage(_senderEmailAddress,email,_smtpServer,
      headers,stream,objectURI, out ID);

   // create and register an async response handler
   AsyncResponseHandler ar = new AsyncResponseHandler(sinkStack);
   SMTPHelper.RegisterAsyncResponseHandler(ID, ar);
}
```

The rest of the SMTPClientTransportSink is just a standard implementation of the mandatory parts of IClientChannelSink. As these methods are not expected to be called for a transport sink, they will either return null or throw an exception.

```
public void AsyncProcessResponse(IClientResponseChannelSinkStack sinkStack,
object state, ITransportHeaders headers, Stream stream)
{
   // not needed in a transport sink!
   throw new NotSupportedException();
}

public Stream GetRequestStream(IMessage msg, ITransportHeaders headers)
{
   // no direct way to access the stream
   return null;
}

public IClientChannelSink NextChannelSink
{
   get
   {
      // no more sinks
      return null;
   }
}
```

The last thing you have to implement before you are able to use this channel is the AsyncResponseHandler class that is used to pass an incoming reply message to the sink stack. Its `HandleAsyncResponsePop3Msg()` method is called by `SMTPHelper.MessageReceived()` whenever the originating call has been placed by using an asynchronous delegate. It calls `SMTPHelper.ProcessMessage()` to split the e-mail message into a Stream object and an ITransportHeaders object and then calls `AsyncProcessResponse()` on the sink stack that has been passed to AsyncResponseHandler in its constructor.

```
using System;
using System.Runtime.Remoting.Channels;
using System.Runtime.Remoting;
using System.IO;
using System.Runtime.Remoting.Messaging;

namespace SmtpChannel
{
   internal class AsyncResponseHandler
   {
      IClientChannelSinkStack _sinkStack;

      internal AsyncResponseHandler(IClientChannelSinkStack sinkStack)
      {
         _sinkStack = sinkStack;
      }

      internal void HandleAsyncResponsePop3Msg(POP3Msg popmsg)
      {
         ITransportHeaders responseHeaders;
         Stream responseStream;
         String ID;

         SMTPHelper.ProcessMessage(popmsg,out responseHeaders,
              out responseStream,out ID);

         _sinkStack.AsyncProcessResponse(responseHeaders,responseStream);
      }
   }
}
```

Well, that's it! You've just completed your first client-side transport channel.

Implementing the Server Channel

As with the client channel, I show you how the server channel will be used before diving into the code. Basically, it looks exactly like the SMTPClientChannel does.

```
<channel
      name="smtpserver"
      type="SmtpChannel.SMTPServerChannel, SmtpChannel"
```

```
      senderEmail="server_1@localhost"
      smtpServer="localhost"
      pop3Server="localhost"
      pop3User="server_1"
      pop3Password="server_1"
      pop3PollInterval="1"
>
```

The parameters for this channel are shown in Table 14-3.

Table 14-3. *Parameters for SMTPServerChannel*

Parameter	Description
name	Unique name for this channel.
senderEmail	The value for the e-mail's From: header. The server will reply to the address specified here.
smtpServer	Your outgoing e-mail server's name.
pop3Server	Your incoming mail server's name.
pop3User	The POP3 user account that is assigned to this client application.
pop3Password	The password for the application's POP3 account.
pop3PollInterval	Interval in seconds at which the framework will check for new mail at the server.

The basic difference between the SMTPClientChannel and the SMTPServerChannel is that the latter registers itself with the POP3PollManager as a server. This means that the POP3Polling instance will constantly check for new e-mails.

The server-side channel has to implement IChannelReceiver, which in turn inherits from IChannel again. These interfaces are shown in Listing 14-5.

Listing 14-5. *IChannel and IChannelReceiver*

```
public interface IChannel
{
    string ChannelName { get; }
    int ChannelPriority { get; }

    string Parse(string url, ref String objectURI);
}

public interface IChannelReceiver: IChannel
{
    object ChannelData { get; }

    string[] GetUrlsForUri(string objectURI);
    void StartListening(object data);
    void StopListening(object data);
}
```

The implementation of SMTPServerChannel itself is quite straightforward. Its constructor checks for the attributes specified in the configuration file and assigns them to local member variables. It also creates a ChannelDataStore object, which is needed for CAOs to communicate back with the server (that is, when creating a CAO using this channel, the server returns the base URL contained in this ChannelDataStore object).

It then creates the sink chain and adds the SMTPServerTransportSink on top of the chain. This is different from the client-side channel, where the constructor only creates a chain of sink providers. This is because on the server side there is only a single sink chain per channel, whereas the client creates a distinct sink chain for each remote object. Finally the constructor calls `StartListening()` to enable the reception of incoming requests.

```
using System;
using System.Collections;
using System.Runtime.Remoting.Channels;
using System.Runtime.Remoting;
using System.Runtime.Remoting.Messaging;

namespace SmtpChannel
{

   public class SMTPServerChannel: BaseChannelWithProperties,
      IChannelReceiver,
      IChannel
   {
      private String _myAddress;
      private String _name;
      private String _smtpServer;
      private String _pop3Server;
      private String _pop3Username;
      private String _pop3Password;
      private int _pop3Pollingtime;

      private SMTPServerTransportSink _transportSink;
      private IServerChannelSinkProvider _sinkProvider;
      private IDictionary _properties;

      private ChannelDataStore _channelData;

      public SMTPServerChannel(IDictionary properties,
             IServerChannelSinkProvider serverSinkProvider)
      {
         _sinkProvider = serverSinkProvider;
         _properties = properties;
         _myAddress = (String) _properties["senderEmail"];
         _name = (String) _properties["name"];
         _pop3Server = (String) _properties["pop3Server"];
         _smtpServer = (String) _properties["smtpServer"];
         _pop3Username = (String) _properties["pop3User"];
```

```
        _pop3Password = (String) _properties["pop3Password"];
        _pop3Pollingtime =
           Convert.ToInt32((String) _properties["pop3PollInterval"]);


        // needed for CAOs!
        String[] urls = { this.GetURLBase() };
        _channelData = new ChannelDataStore(urls);

        // collect channel data from all providers
        IServerChannelSinkProvider provider = _sinkProvider;
        while (provider != null)
        {
           provider.GetChannelData(_channelData);
           provider = provider.Next;
        }

        // create the sink chain
        IServerChannelSink snk =
           ChannelServices.CreateServerChannelSinkChain(_sinkProvider,this);

        // add the SMTPServerTransportSink as a first element to the chain
        _transportSink = new SMTPServerTransportSink(snk, _smtpServer,
           _myAddress);

        // start to listen
        this.StartListening(null);
     }
```

The constructor calls `GetURLBase()`, which provides a way for this channel to return its base URL.

```
private String GetURLBase()
{
   return "smtp:" + _myAddress;
}
```

You also have to implement IChannel's methods and properties: `Parse()`, `ChannelName`, and `ChannelPriority`. The implementation itself looks exactly the same as it did for the client-side channel.

```
public string Parse(string url, out string objectURI)
{
   String email;
   SMTPHelper.parseURL(url, out email, out objectURI);
   if (email == null || email=="" || objectURI == null || objectURI =="")
   {
      return null;
   }
   else
```

```
    {
        return "smtp:" + email;
    }
}

public string ChannelName
{
    get
    {
        return _name;
    }
}

public int ChannelPriority
{
    get
    {
        return 0;
    }
}
```

The single most important method of a server-side channel is `StartListening()`. Only after it is called will the server be able to receive requests and to handle them.

In the SMTPServerChannel, this method registers its connection as a server with the POP3PollManager. It next registers the server-side transport sink and its e-mail address with the SMTPHelper. This last step will enable the helper to dispatch requests based on the destination e-mail address.

```
public void StartListening(object data)
{
    // register the POP3 account for polling
    POP3PollManager.RegisterPolling(_pop3Server,_pop3Username,
        _pop3Password,_pop3Pollingtime,true);

    // register the e-mail address as a server
    SMTPHelper.RegisterServer(_transportSink,_myAddress);
}

public void StopListening(object data)
{
    // not needed ;-)
}
```

To enable CAOs to work correctly, you must implement the method `GetUrlsForUri()` and the property `ChannelData`. The first allows the framework to convert a given object's URI into a complete URL (including the protocol-specific part, such as smtp:user@host.com). The second returns the channel data object that is used by the framework to provide the complete URL for a client-activated object.

```
public string[] GetUrlsForUri(string objectURI)
{
   String[] urls;
   urls = new String[1];
   if (!(objectURI.StartsWith("/")))
      objectURI = "/" + objectURI;

   urls[0] = this.GetURLBase() + objectURI;
   return urls;
}

public object ChannelData
{
   get
   {
      return _channelData;
   }
}
```

Creating the Server's Sink

A server-side transport sink has to implement IServerChannelSink, which in turn extends IChannelSinkBase. You might already know the interfaces shown in Listing 14-6 from Chapter 13, where they were used to extend the remoting infrastructure.

Listing 14-6. *IChannelSinkBase and IServerChannelSink*

```
public interface IChannelSinkBase
{
    IDictionary Properties { get; }
}

public interface IServerChannelSink : IChannelSinkBase
{
    IServerChannelSink NextChannelSink { get; }

    ServerProcessing ProcessMessage(IServerChannelSinkStack sinkStack,
           IMessage requestMsg, ITransportHeaders requestHeaders,
           Stream requestStream, ref IMessage responseMsg,
           ref ITransportHeaders responseHeaders, ref Stream responseStream);

    void AsyncProcessResponse(IServerResponseChannelSinkStack sinkStack,
           object state, IMessage msg, ITransportHeaders headers,
           Stream stream);

    Stream GetResponseStream(IServerResponseChannelSinkStack sinkStack,
           object state, IMessage msg, ITransportHeaders headers);

}
```

The implementation of SMTPServerTransportSink is a little bit different from classic channel sinks. First and foremost, it is not created by a sink provider but instead directly by the channel that passes the reference to the next sink, the SMTP server's address and the server's own address to the newly created SMTPServerTransportSink.

Additionally, you'll need a private class to hold state information about the origin of the request and its message ID to process the asynchronous replies.

```
using System;
using System.IO;
using System.Runtime.Remoting.Channels;
using System.Runtime.Remoting;
using System.Runtime.Remoting.Messaging;

namespace SmtpChannel
{

   public class SMTPServerTransportSink: IServerChannelSink

   {
      // will be used as a state object for the async reply
      private class SMTPState
      {
         internal String ID;
         internal String responseAddress;
      }

      private String _smtpServer;
      private String _myAddress;
      private IServerChannelSink _nextSink;

      public SMTPServerTransportSink(IServerChannelSink nextSink,
            String smtpServer, String myAddress)
      {
         _nextSink = nextSink;
         _smtpServer =smtpServer;
         _myAddress = myAddress;
      }
```

One of the main differences between a server-side transport sink and a "conventional" channel sink is that the latter receives its parameters via the `ProcessMessage()` method. The transport sink instead does not define a specific way to receive the incoming request from the underlying transport mechanisms.

In the case of the SMTPServerTransportSink, it receives a POP3Msg object from the SMTPHelper and processes it in `HandleIncomingMessage()`. It first splits the e-mail message into a Stream object and an ITransportHeaders object. It then creates a new state object of type SMTPState and populates its properties from the e-mail's values.

Next, it creates a ServerChannelSinkStack and pushes itself and the newly created state object onto it before handing over the processing to the next sink in its chain. When this method is finished, it returns a ServerProcessing value. This indicates whether the message has been handled synchronously, asynchronously, or as a one-way message.

The SMTPServerTransportSink now behaves accordingly. If the request has been handled synchronously, it generates and sends a response message. For asynchronous calls, it waits for the framework to call its AsyncProcessResponse() method. For one-way calls, it does nothing at all.

```
public void HandleIncomingMessage(POP3Msg popmsg)
{
   Stream requestStream;
   ITransportHeaders requestHeaders;
   String ID;

   // split the message in Stream and ITransportHeaders
   SMTPHelper.ProcessMessage(popmsg,out requestHeaders,
      out requestStream, out ID);

   // create a new sink stack
   ServerChannelSinkStack stack = new ServerChannelSinkStack();

   // create a new state object and populate it
   SMTPState state = new SMTPState();
   state.ID = ID;
   state.responseAddress = SMTPHelper.GetCleanAddress(popmsg.From );

   // push this sink onto the stack
   stack.Push(this,state);

   IMessage responseMsg;
   Stream responseStream;
   ITransportHeaders responseHeaders;

   // forward the call to the next sink
   ServerProcessing proc = _nextSink.ProcessMessage(stack,null,requestHeaders,
            requestStream, out responseMsg, out responseHeaders,
            out responseStream);

   // check the return value
   switch (proc)
   {
      // this message has been handled synchronously
      case ServerProcessing.Complete:
         // send a response message
         SMTPHelper.SendResponseMessage(_myAddress,
            state.responseAddress,_smtpServer,responseHeaders,
            responseStream,state.ID);
         break;
```

```
      // this message has been handled asynchronously
      case ServerProcessing.Async:
         // nothing needs to be done yet
         break;

      // it's been a one way message
      case ServerProcessing.OneWay:
         // nothing needs to be done yet
         break;
   }
}
```

`AsyncProcessResponse()` is called when the framework has completed the execution of an underlying asynchronous method. The SMTPServerTransportSink in this case generates a response message and sends it to the client.

```
public void AsyncProcessResponse(
   IServerResponseChannelSinkStack sinkStack, object state,
   IMessage msg, ITransportHeaders headers, System.IO.Stream stream)
{

   // fetch the state object
   SMTPState smtpstate = (SMTPState) state;

   // send the response e-mail
   SMTPHelper.SendResponseMessage(_myAddress,
      smtpstate.responseAddress,_smtpServer,headers,
      stream,smtpstate.ID);
}
```

What's still left in SMTPServerTransportSink is the implementation of the other mandatory methods and properties defined in IServerChannelSink. Most of them will not be called for a transport sink and will therefore only return null or throw an exception.

```
public IServerChannelSink NextChannelSink
{
   get
   {
      return _nextSink;
   }
}

public System.Collections.IDictionary Properties
{
   get
   {
      // not needed
      return null;
   }
}
```

```
public ServerProcessing ProcessMessage(
   IServerChannelSinkStack sinkStack, IMessage requestMsg,
   ITransportHeaders requestHeaders, Stream requestStream,
   out IMessage responseMsg, out ITransportHeaders responseHeaders,
   out Stream responseStream)
{
   // will never be called for a server side transport sink
   throw new NotSupportedException();
}

public Stream GetResponseStream(
   IServerResponseChannelSinkStack sinkStack, object state,
   IMessage msg, ITransportHeaders headers)
{
   // it's not possible to directly access the stream
   return null;
}
```

Great! You have now finished implementing your own transport channel!

Wrapping the Channel

As you've seen with the default .NET Remoting channels, you don't have to manually create and register HttpClientChannel and HttpServerChannel but can instead use the combination in the form of HttpChannel. This isn't strictly needed for compatibility with the .NET Remoting framework, but it does provide more comfort for the developers using this channel. An additional feature you might want to implement is the default assignment of a formatter to this channel. I'm now going to show you how to do this to create an SmtpChannel class.

First, the combined channel has to extend BaseChannelWithProperties and implement IChannelSender and IChannelReceiver. Nevertheless, there won't be too much logic in SmtpChannel, as it will delegate most of its work to either SMTPClientChannel or SMTPServerChannel.

To check if an application wants to act as a server for .NET Remoting requests via SMTP, you have to introduce another attribute that can be used in the configuration file: `isServer`. When this is set to "yes", the SmtpChannel will create an SMTPServerChannel as well; otherwise it will only create an SMTPClientChannel.

The SmtpChannel has to implement a different constructor that allows the framework to pass both an IClientChannelSinkProvider *and* an IServerChannelSinkProvider object to it. It will then check whether either of these is `null` and create a default SOAP formatter in this case.

All other methods that have to be implemented to support the specified interfaces will just forward their calls to the respective client or server channel. This is shown in Listing 14-7.

Listing 14-7. *The SmtpChannel*

```
using System;
using System.Collections;
using System.Runtime.Remoting.Channels;
using System.Runtime.Remoting;
using System.Runtime.Remoting.Messaging;
```

```
namespace SmtpChannel
{
   public class SmtpChannel: BaseChannelWithProperties,
      IChannelSender, IChannelReceiver
   {
      SMTPClientChannel _clientchannel;
      SMTPServerChannel _serverchannel;
      String _name;

      public SmtpChannel (IDictionary properties,
           IClientChannelSinkProvider clientSinkProvider,
           IServerChannelSinkProvider serverSinkProvider)
      {
         if (clientSinkProvider == null)
         {
            clientSinkProvider = new SoapClientFormatterSinkProvider();
         }

         // create the client channel
         _clientchannel = new SMTPClientChannel(properties, clientSinkProvider);

         if ((properties["isServer"] != null) &&
            ((String) properties["isServer"] == "yes" ))
         {
            if (serverSinkProvider == null)
            {
               serverSinkProvider = new SoapServerFormatterSinkProvider();
            }

            // create the server channel
            _serverchannel = new SMTPServerChannel( properties,
                  serverSinkProvider);
         }

         _name = (String) properties["name"];
      }

      public IMessageSink CreateMessageSink(string url,
         object remoteChannelData, out string objectURI)
      {
         return _clientchannel.CreateMessageSink(url,
            remoteChannelData, out objectURI);
      }

      public string Parse(string url, out string objectURI)
      {
         return _clientchannel.Parse(url, out objectURI);
      }
```

```
public string ChannelName
{
   get
   {
      return _name;
   }
}

public int ChannelPriority
{
   get
   {
      return 0;
   }

}

public void StartListening(object data)
{
   if (_serverchannel != null)
   {
      _serverchannel.StartListening(data);
   }
}

public void StopListening(object data)
{
   if (_serverchannel != null)
   {
      _serverchannel.StopListening(data);
   }
}

public string[] GetUrlsForUri(string objectURI)
{
   if (_serverchannel != null)
   {
      return _serverchannel.GetUrlsForUri(objectURI);
   }
   else
   {
      return null;
   }
}

public object ChannelData
{
   get
```

```
        {
           if (_serverchannel != null )
           {
              return _serverchannel.ChannelData;
           }
           else
           {
              return null;
           }
        }
     }

  }
}
```

Using the SmtpChannel

What you've seen previously constitutes a full-featured transport channel. It supports every .NET Remoting functionality: synchronous calls, asynchronous calls, and event notification. In addition, client-activated objects can be used with this channel. To use it on the server side, you can register it by implementing a configuration file like this:

```
<configuration>
  <system.runtime.remoting>
    <application>
      <channels>
         <channel name="smtp"
            type="SmtpChannel.SmtpChannel, SmtpChannel"
            senderEmail="server_1@localhost"
            smtpServer="localhost"
            pop3Server="localhost"
            pop3User="server_1"
            pop3Password="server_1"
            pop3PollInterval="1"
            isServer="yes" />
      </channels>

      <service>
        <wellknown mode="Singleton"
                   type="Service.SomeSAO, Service"
                   objectUri="SomeSAO.soap" />
      </service>
    </application>
  </system.runtime.remoting>
</configuration>
```

The corresponding client-side configuration file might look like this:

```
<configuration>
  <system.runtime.remoting>
    <application>

     <channels>
        <channel name="smtp"
            type="SmtpChannel.SmtpChannel, SmtpChannel"
            senderEmail="client_1@localhost"
            smtpServer="localhost"
            pop3Server="localhost"
            pop3User="client_1"
            pop3Password="client_1"
            pop3PollInterval="1"
            isServer="yes" />
      </channels>

      <client>
        <wellknown type="Service.SomeSAO, Service"
                   url="smtp:server_1@localhost/SomeSAO.soap" />
      </client>

      <client url="smtp:server_2@localhost">
        <activated type="Service.SomeCAO, Service" />
      </client>

    </application>
  </system.runtime.remoting>
</configuration>
```

In the source code download that accompanies this book online, you'll find not only the complete implementation of this channel, but also a test environment consisting of three projects (two servers and a client) that shows the following features using the SmtpChannel:

- Server-activated objects
- Client-activated objects
- Synchronous calls
- Asynchronous calls using a delegate
- Raising and handling events
- Passing references to CAOs between different applications

Preparing Your Machine

To run the samples on your machine, you'll need to have access to three e-mail accounts (two for the servers and one for the client) with SMTP and POP3. For testing purposes I therefore recommend that you download and install Mercury/32, a free e-mail server, to allow you to easily perform the configuration without having to bother any system administrators. You can get it from `http://www.pmail.com`.

Please create three user accounts in Mercury/32 after installing and running it (Configuration ➤ Manage local users), each having the same password as the user name: client_1, server_1, and server_2. You can see the final state in Figure 14-1.

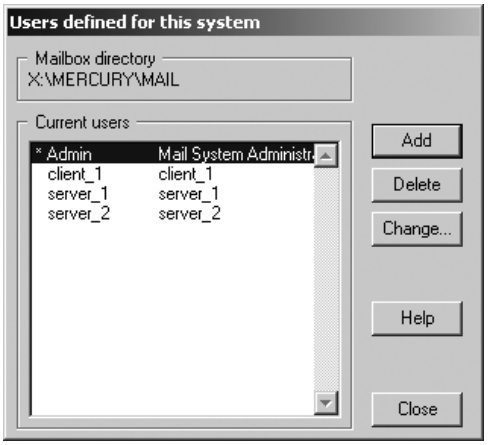

Figure 14-1. *These user accounts are needed for this example.*

You also need to change to Mercury core configuration (Configuration ➤ Mercury core module) to recognize the local domain. To do this, switch to the Local domains tab and enter **localhost** in the Local host or server and Internet name text boxes, as shown in Figure 14-2.

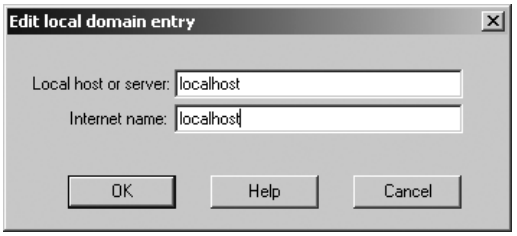

Figure 14-2. *Preparing the core modules*

Some Final Words of Caution

Developing a custom transport channel is a nontrivial task. In some cases, when you have to work with different protocols—for example, when talking with embedded devices—it might be easier to just implement a raw socket or serial port connection directly as an application-specific API library.

Developing a transport channel should be a very conscious choice, for example, if you'd like to allow interoperability with a third-party distributed application technology. As you will see in the last chapter of this book, there are already a number of bridges between .NET and Java or between .NET and CORBA that are based on custom transport channels. If you are about to decide to create a custom channel just to interoperate between these platforms, it might be easier to look at one of the already existing solutions.

Apart from this, the same disclaimer I've used at the end of the previous chapter applies here as well: you should take explicit care not to tie your client- or server-side business logic too much to the .NET Remoting framework. If your business logic code depends on side effects introduced by your custom channel, it will be very hard to migrate it to future technology. Always remember: business code often outlives its initial environment, but you have to plan accordingly to allow for future changes.

Summary

After reading this chapter, you've finally reached the level of .NET Remoting wizardship. Not only do you know how to extend the framework using custom sinks and providers, but now you can also implement a complete channel from scratch. You learned that most work in implementing a custom channel has to be expended in understanding and encapsulating the underlying protocol. Mapping an asynchronous protocol to synchronous calls and vice versa is an especially important and challenging task. You also know how to implement IChannelSender and IChannelReceiver, how to combine them into a top-level channel, and how to assign default formatters to a channel.

In the next chapter, I introduce you to the possibilities of using the basic principle of .NET Remoting, which is the processing of method calls using messages instead of stack-based calling conventions in your local applications.

CHAPTER 15

Context Matters

This chapter is about message-based processing in local applications. Here you learn how you can intercept calls to objects to route them through IMessageSinks. This routing allows you to create and maintain parts of your application's business logic at the metadata level by using attributes. You also discover why it might be a good idea to do so.

Caution Everything in this chapter is 100 percent undocumented. Reliance on these techniques is not supported by either Microsoft, the publisher, or the author of this book. Use at your own risk! If your computer won't work afterwards, your toaster blows up, or your car doesn't start, I assume no liability whatsoever. You're now about to enter the uncharted territories of .NET and you do so at your own risk. I can only provide some guidance.

Well, it's great that you're still with me after this introductory warning. So let's start with a look at some common business applications. You will quite likely have some object model that holds local data before it's committed to the database. Those classes will contain parts of your business logic. For example, assume that your application provides an instant way for employees of your company to donate various amounts of their paychecks to charity organizations. In this case, you might have a data object that looks like the one shown in Listing 15-1, which allows a user to set an organization's name and the donation of a specified amount to it.

Listing 15-1. *The First Version of the Organization Object*

```
using System;

namespace ContextBound
{
   public class Organization
   {
      String _name;
      double _totalDonation;

      public String Name
      {
         set
         {
            _name = value;
         }
         get
         {
            return _name;
         }
      }

      public void Donate(double amount)
      {
         _totalDonation = _totalDonation + amount;
      }

   }
}
```

You might also have some database restriction or business logic that limits an organization's name to 30 characters and allows a maximum donation of $100.00. Therefore, you need to extend `Donate()` and the setter of `Name` to check for this logic.

```
public String Name
{
   set
   {
      if (value != null && value.Length > 30)
      {
         throw new Exception("This field must not be longer than 30 characters");
      }

      _name = value;
   }
```

```
   get
   {
      return _name;
   }
}

public void Donate(double amount)
{
   if (amount > 100)
   {
      throw new Exception("This parameter must not be greater than 100.");
   }
   _totalDonation = _totalDonation + amount;
}
```

You're checking the business logic and your application works as expected. So far, so good. The problems only commence as soon as more developers start using your objects as the base for their applications, because they don't discover about those restrictions by reading the interface definition alone. As in most real-world applications, the business logic is in this case hidden inside the implementation and is not part of the metadata level. There is no way for another developer to tell that the maximum amount for a valid donation is $100.00 without looking at your source code.

If you're a well-informed developer, you already know that you can at least document those parameters using inline XML comments to automatically generate online documentation for your classes—but you still have to document and implement the logic in two separate places. If you've never, ever changed any implementation detail without updating the inline documentation, you don't need to read further—you've already solved the problem.

Working at the MetaData Level

In most projects though (at least in some I've recently heard of), there is a direct proportionality between days to deadline and quality of documentation. Somehow people tend to forget to update comments as soon as their boss is reminding them that they should have shipped it yesterday.

Wouldn't it be great to just specify those checks using some source code attributes and have some "black magic" happen between the client and your objects that takes care of checking the passed values against those attributes?

In a perfect world, these methods might simply look like this:

```
public String Name
{
   [Check(MaxLength=30)]
   set
   {
      _name = value;
   }
```

```
   get
   {
      return _name;
   }
}

public void Donate([Check(MaxValue=100)] double amount)
{
   _totalDonation = _totalDonation + amount;
}
```

Now the documentation of your business logic is applied on the metadata level! You could easily use reflection to generate printed or online documentation that includes these basic business logic checks as well.

Well, unfortunately, no checks have been done yet. In fact, when using this class, you could easily set `Name` to any possible value and `Donate()` to whatever amount you'd like.

■ **Caution** You're now really about to read about unsupported and undocumented features of .NET Framework. Your mileage may vary.

What's still missing is that magic something I mentioned that would sit between the client and your object (running maybe within the same process and application) and perform those checks. This is where ContextBoundObject enters the game.

Creating a Context

When you create a class that is derived from ContextBoundObject, nothing special happens yet: by default all objects are still created in the same context. You can, however, decorate this class with an attribute that inherits from ContextAttribute and overrides the following two methods:

```
public bool IsContextOK(Context ctx, IConstructionCallMessage ctor)
public void GetPropertiesForNewContext(IConstructionCallMessage ctor)
```

When doing this, the first method is called whenever someone is creating a new instance of the target class (for example, the previous Organization class). If it returns true, nothing happens, and the object is created in the same context as the client. There won't be the chance to intercept a call from the client to this instance by using a message sink.

If the method returns false, on the other hand, a new "virtual" remoting boundary, the context, is created. In this case, the framework will subsequently call `GetPropertiesForNewContext()` to allow you to add the IContextProperty objects that you want to use with this context.

The implementation of a complete attribute that will later be used to create a sink to intercept calls to this object is shown in Listing 15-2.

Listing 15-2. *A ContextAttribute That Allows You to Intercept Calls*

```
using System;
using System.Runtime.Remoting;
using System.Runtime.Remoting.Contexts;
using System.Runtime.Remoting.Activation;
using System.Runtime.Remoting.Messaging;

namespace ContextBound
{
   [AttributeUsage(AttributeTargets.Class)]
   public class CheckableAttribute: ContextAttribute
   {
      public CheckableAttribute(): base ("MyInterception") { }

      public override bool IsContextOK(Context ctx,
         IConstructionCallMessage ctor)
      {
         // if this is already an intercepting context, it's ok for us
         return ctx.GetProperty("Interception") != null;
      }

      public override void GetPropertiesForNewContext(
         IConstructionCallMessage ctor)
      {
         // add the context property that will later create a sink
         ctor.ContextProperties.Add(new CheckableContextProperty());
      }
   }
}
```

An IContextProperty on its own doesn't provide you with a lot of functionality, as you can see in Listing 15-3.

Listing 15-3. *The IContextProperty Interface*

```
public interface IContextProperty
{
    string Name { get; }

    void Freeze(Context newContext);
    bool IsNewContextOK(Context newCtx);
}
```

`Freeze()` is called when the context itself is frozen. This indicates that no change of context properties is allowed afterwards. `IsNewContextOk()` is called after all context attributes have added their context properties to allow your property to check for dependencies. If IContextProperty A can only be used together with IContextProperty B, it can check in this method if both properties are available for the newly created context. If this method returns `false`, an exception will be thrown.

Name simply has to return the context property's name that will be used to retrieve it by calling Context.GetProperty("<name>"). To be able to create a sink to intercept calls to this object, this class will have to implement one of the following interfaces: IContributeObjectSink, IContributeEnvoySink, IContributeClientContextSink, or IContributeServerContextSink. In the examples to follow, I use IContributeObjectSink, which is shown in Listing 15-4.

Listing 15-4. *The IContributeObjectSink Interface*

```
public interface IContributeObjectSink
{
    IMessageSink GetObjectSink(MarshalByRefObject obj, IMessageSink nextSink);
}
```

To create a new instance of CheckerSink, you can implement the IContextProperty, as shown in Listing 15-5.

Listing 15-5. *The CheckableContextProperty*

```
using System;
using System.Runtime.Remoting;
using System.Runtime.Remoting.Contexts;
using System.Runtime.Remoting.Activation;
using System.Runtime.Remoting.Messaging;

namespace ContextBound
{

   public class CheckableContextProperty: IContextProperty,
      IContributeObjectSink
   {
      public bool IsNewContextOK(Context newCtx)
      {
         return true;
      }

      public void Freeze(Context newContext)
      {
         // nothing to do
      }

      public string Name
      {
         get
         {
            return "Interception";
         }
      }
```

```
      public IMessageSink GetObjectSink(MarshalByRefObject obj,
            IMessageSink nextSink)
      {
         return new CheckerSink(nextSink);
      }

   }
}
```

CheckerSink itself is a common IMessageSink implementation. Its first iteration is shown in Listing 15-6.

Listing 15-6. *The CheckerSink's First Iteration*

```
using System;
using System.Reflection;
using System.Runtime.Remoting;
using System.Runtime.Remoting.Activation;
using System.Runtime.Remoting.Contexts;
using System.Runtime.Remoting.Messaging;

namespace ContextBound
{

   public class CheckerSink: IMessageSink
   {
      IMessageSink _nextSink;

      public CheckerSink(IMessageSink nextSink)
      {
         _nextSink = nextSink;
      }

      public IMessage SyncProcessMessage(IMessage msg)
      {
         Console.WriteLine("CheckerSink is intercepting a call");
         return _nextSink.SyncProcessMessage(msg);
      }

      public IMessageCtrl AsyncProcessMessage(IMessage msg,
         IMessageSink replySink)
      {
         Console.WriteLine("CheckerSink is intercepting an async call");
         return _nextSink.AsyncProcessMessage(msg,replySink);
      }
```

```
      publicIMessageSink NextSink
      {
         get
         {
            return _nextSink;
         }
      }
   }
}
```

To enable this way of intercepting the Organization class shown at the beginning of this chapter, you have to mark it with [Checkable] and have it inherit from ContextBoundObject to create the context property.

The Organization class, which is shown in Listing 15-7, does not yet employ the use of custom attributes for checking the maximum amount of a single donation or the maximum length of the organization's name. It just demonstrates the basic principle of interception.

Listing 15-7. *The Organization Is Now a ContextBoundObject*

```
using System;

namespace ContextBound
{
   [Checkable]
   public class Organization: ContextBoundObject
   {
      String _name;
      double _totalDonation;

      public String Name
      {
         set
         {
            _name = value;
         }
         get
         {
            return _name;
         }
      }

      public void Donate(double amount)
      {
         Organization x = new Organization();
         x.Name = "Hello World";
         _totalDonation = _totalDonation + amount;
      }

   }
}
```

A simple client for this class is shown in Listing 15-8.

Listing 15-8. *This Client Is Using the ContextBoundObject*

```
using System;
using System.Runtime.Remoting.Contexts;

namespace ContextBound
{
   public class TestClient
   {
      public static void Main(String[] args) {
         Organization org = new Organization();
         Console.WriteLine("Will set the name");
         org.Name = "Happy Hackers";
         Console.WriteLine("Will donate");
         org.Donate(103);

         Console.WriteLine("Finished, press <return> to quit.");
         Console.ReadLine();
      }
   }
}
```

When this application is started, you will see the output shown in Figure 15-1.

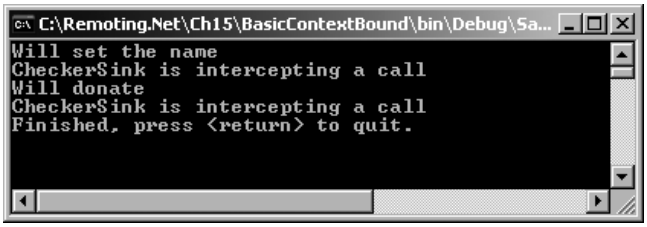

Figure 15-1. *The application's output when using the ContextBoundObject*

As you can see here, the CheckerSink intercepts the setting of the property `Name` and the call to `Donate()`, although it doesn't yet do anything to check the constraints I mentioned earlier.

The first step to enabling the sink to do something useful is to create a custom attribute that will later be used to designate a parameter's maximum length and maximum value. This attribute, which can be used for parameters and methods, stores the properties `MaxLength`, `MaxValue`, and `NonNull`, as shown in Listing 15-9. Its `DoCheck()` method will later be called by the sink to check a given value against the attribute's definition.

Listing 15-9. *The CheckAttribute*

```
using System;

namespace ContextBound
{
   [AttributeUsage (AttributeTargets.Parameter | AttributeTargets.Method)]
   public class CheckAttribute: Attribute
   {
      private int _maxLength;
      private int _maxValue;
      private bool _nonNull;

      public int MaxLength {
         get {
            return _maxLength;
         }
         set {
            _maxLength = value;
         }
      }

      public int MaxValue
      {
         get
         {
            return _maxValue;
         }
         set
         {
            _maxValue = value;
         }
      }

      public bool NonNull
      {
         get
         {
            return _nonNull;
         }
         set
         {
            _nonNull = value;
         }
      }
```

```
      public void DoCheck (Object val)
      {
         // check for NonNull
         if (_nonNull && val == null)
         {
            throw new Exception("This value must not be null");
         }

         // check for MaxLength
         if (_maxLength > 0 && val.ToString().Length > _maxLength)
         {
            throw new Exception("This value must not be longer than " +
                     _maxLength + " characters");
         }

         // check for MaxValue
         if (_maxValue > 0)
         {
            if ((double) val > _maxValue)
            {
               throw new Exception("This value must not be higher than " +
                     _maxValue );
            }

         }
      }
   }
}
```

To make use of this attribute in the Organization class, you have to mark the parameter to `Donate()` and the set method for the `Name` property, as shown here:

```
public String Name
{
   [Check(NonNull=true,MaxLength=30)]
   set
   {
      _name = value;
   }
   get
   {
      return _name;
   }
}

public void Donate([Check(NonNull=true,MaxValue=100)] double amount)
{
   _totalDonation = _totalDonation + amount;
}
```

Checking Parameters in an IMessageSink

Listing 15-10 shows the implementation of the CheckerSink. Calls from `SyncProcessMessage()` and `AsyncProcessMessage()` have been added to the private `DoCheck()` method, which iterates over the assigned attributes and forwards the business logic checks to `CheckAttribute.DoCheck()` for each parameter that is marked with this attribute.

Listing 15-10. *The CheckerSink*

```
using System;
using System.Reflection;
using System.Runtime.Remoting;
using System.Runtime.Remoting.Activation;
using System.Runtime.Remoting.Contexts;
using System.Runtime.Remoting.Messaging;

namespace ContextBound
{

   public class CheckerSink: IMessageSink
   {
      IMessageSink _nextSink;
      String _mType;
      public CheckerSink(IMessageSink nextSink, String mType)
      {
         _nextSink = nextSink;
         _mType = mType;
      }

      public IMessage SyncProcessMessage(IMessage msg)
      {
         DoCheck(msg);
         return _nextSink.SyncProcessMessage(msg);
      }

      public IMessageCtrl AsyncProcessMessage(IMessage msg,
         IMessageSink replySink)
      {
         DoCheck(msg);
         return _nextSink.AsyncProcessMessage(msg,replySink);
      }

      public IMessageSink NextSink
      {
         get
         {
            return _nextSink;
         }
      }
```

```
private void DoCheck(IMessage imsg)
{
   // not interested in IConstructionCallMessages
   if (imsg as IConstructionCallMessage != null)  return;

   // but only interested in IMethodMessages
   IMethodMessage msg = imsg as IMethodMessage;
   if (msg == null) return;


   // check for the Attribute
   MemberInfo methodbase = msg.MethodBase;

   object[] attrs = methodbase.GetCustomAttributes(false);

   foreach (Attribute attr in attrs)
   {
      CheckAttribute check = attr as CheckAttribute;

      // only interested in CheckAttributes
      if (check == null) continue;

      // if the method only has one parameter, place the check directly
      // on it (needed for property set methods)
      if (msg.ArgCount == 1)
      {
         check.DoCheck(msg.Args[0]);
      }
   }

   // check the Attribute for each parameter of this method
   ParameterInfo[] parms = msg.MethodBase.GetParameters();

   for (int i = 0;i<parms.Length;i++)
   {
      attrs = parms[i].GetCustomAttributes(false);
      foreach (Attribute attr in attrs)
      {
         CheckAttribute check = attr as CheckAttribute;

         // only interested in CheckAttributes
         if (check == null) continue;

         // if the method only has one parameter, place the check directly
         // on it (needed for property set methods)
```

```
                check.DoCheck(msg.Args[i]);
            }
        }

    }

  }
}
```

You can then change the sample client to demonstrate what happens when it performs an invalid operation, as shown in Listing 15-11.

Listing 15-11. *This Client Does Not Honor the Business Logic Constraints*

```
using System;
using System.Runtime.Remoting.Contexts;

namespace ContextBound
{
   public class TestClient
   {
      public static void Main(String[] args) {
         Organization org = new Organization();
         try
         {
            Console.WriteLine("Will set the name");
            org.Name = "Happy Hackers";
            Console.WriteLine("Will donate");
            org.Donate(99);
            Console.WriteLine("Will donate more");
            org.Donate(102);
         }
         catch (Exception e)
         {
            Console.WriteLine("Exception: {0}",e.Message);
         }

         Console.WriteLine("Finished, press <return> to quit. ");
         Console.ReadLine();
      }
   }
}
```

When you start this application, you will get the output shown in Figure 15-2.

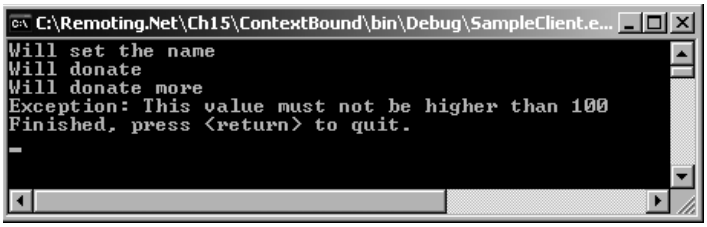

Figure 15-2. *The client's illegal operation is prohibited by the CheckerSink.*

Great! You are now checking your business logic constraints by using attributes that are assigned at the metadata level instead of checks that are hidden in your source code.

One interesting One interesting consideration that I have not yet mentioned is the following: what would happen if the first Organization object instantiates another Organization object and calls the `Donate()` method on the secondary object? Will this call also go through the message sink? In fact, in the current configuration it won't. This example just protects your class library from "outside" clients but doesn't affect any calls inside this context. This is because the CheckableAttribute's `IsContextOK()` only requests a new context when it's called from outside a checked context.

To make *all* calls to Organization (no matter what their origin) go through the CheckerSink, you'd have to change CheckableAttribute to return false from `IsContextOK()`:

```
public override bool IsContextOK(Context ctx, IConstructionCallMessage ctor)
{
    return false;
}
```

This will request a new context for each and every instance of any class that is marked with `[Checkable]` and that inherits from ContextBoundObject.

Summary

In this last chapter, I showed you some undocumented techniques to move constraints away from the implementation up to the metadata level. When using this approach together with reflection on the types of your class library, you will be able to automatically generate documentation that includes all those metadata-level checks. You learned about using different contexts in your local application and how to use IContextProperty and IContributeObjectSink to intercept calls to your objects by using IMessageSink objects.

I just want to remind you that context sinks are a great technology, but unfortunately not yet officially supported or documented by Microsoft. If you use them, it will be at your own risk! If any problems occur when doing so, you will be on your own. But isn't that the fate of anyone who's going to enter uncharted territory?

Conclusion

In this book you learned .NET Remoting from the basics to very advanced topics. In the first chapters, I introduced you to the various kinds of remote objects and how to create and register them. I covered the intricacies of client-activated objects and server-activated objects. You also learned about the various ways of generating the necessary metadata to allow the .NET Remoting framework to create transparent proxies. I showed you the deployment options for remoting servers that can be either managed applications (including console applications, Windows services, and Windows Form applications) and IIS. I then showed you more advanced topics such as security, event handling, versioning, and lifetime management by using leases and sponsors.

In the second part of the book, I showed you how .NET Remoting works internally. You were introduced to proxies, messages, transport channels, formatters, message sinks, and channel sinks. After covering those architectural basics, I showed you how to leverage the .NET Remoting framework's extensibility model by implementing your own sinks and sink providers. At the end of the second part, you finally learned how to implement a complete transport channel from scratch and how to use ContextBoundObject to intercept message calls.

You are now well prepared for the development of distributed applications using the .NET Framework—so go ahead and do your stuff!

PART 3

Reference

APPENDIX A

.NET Remoting Usage Reference

The core classes for .NET Remoting reside in the System.Runtime.Remoting namespace. In this appendix, you will find a reference of the types that are usually used when writing .NET Remoting applications and that were used in the first part of the book. I will not explain any classes, interfaces, and enumerations used for extending the .NET Remoting infrastructure in this appendix. The reference for extensibility can be found in Appendix B, which presents the types you use for extending the .NET Remoting infrastructure.

Many entries are cross-referenced with relevant chapters as well as entries to the MSDN documentation where appropriate. MSDN documentation will include further details in many cases. I will also cover several classes from the following subnamespaces:

- System.Runtime.Remoting.Channels
- System.Runtime.Remoting.Lifetime
- System.Runtime.Remoting.Messaging
- System.Runtime.Remoting.Metadata
- System.Runtime.Remoting.Services
- System.Runtime.Serialization

But before starting with the types defined in these namespaces, I have to cover a handful of types defined in other namespaces, but used by the .NET Remoting infrastructure, which are used in the chapters throughout the book.

System Types

Here I'll explain some types that are not defined in the Remoting namespace but are used by the .NET Remoting infrastructure. Often you won't be using the types directly, but it's important to have a brief understanding of these types.

System.Activator Class

The Activator class is primarily used for either creating instances of new objects locally or remotely or obtaining references to existing instances of remote objects. The `CreateInstance()` and `CreateInstanceFrom()` methods can be used for creating new instances of .NET-based types, whereas the `CreateComInstanceFrom()` method obtains a COM object name (in the format "library.classname") as parameter for creating registered COM objects.

In the examples in this book, you also saw the `GetObject()` method used for retrieving a remote well-known object that already exists on a .NET Remoting server.

Usage example:

```
MyClass MyFirst = (MyClass)Activator.CreateInstance(typeof(MyClass));

ObjectHandle handle = Activator.CreateInstance(
                                "MyAssembly",
                                "MyNamespace.MyClass",
                                new object[] {"First", 2});
MyClass MySecond = (MyClass)handle.Unwrap();

IRemoteComponent MyThird = (IRemoteComponent)Activator.GetObject(
                                    typeof(IRemoteComponent),
                                    "tcp://localhost:8080/MyRemote.rem");
```

The use of the Activator class is demonstrated in many of the chapters of the book, the first time being within the first .NET Remoting example, which appears in Chapter 2.

More information on MSDN:
`http://msdn.microsoft.com/library/en-us/cpref/html/frlrfsystemactivatorclasstopic.asp`

System.MarshalByRefObject Class

The MarshalByRefObject class usually is not used directly by the application programmer. This class enables access to objects living in other application domains. Such objects can reside in application domains of the same or other processes on the same machine or in processes on remote machines.

Objects that need to be accessible remotely must be inherited from MarshalByRefObject.

Usage example:

```
public class MyClass : MarshalByRefObject, IRemoteComponent
{
  public void Foo()
  {
    // do something here
    // can be called remotely because
    // MyClass inherits from MarshalByRefObject
  }
}
```

As all remoteable objects have to be inherited from MarshalByRefObject, I am using this class as a base class throughout all the chapters of the book for such objects. The first occurrence and explanation can be found in Chapter 2 when creating the first .NET Remoting example.

More information on MSDN:
http://msdn.microsoft.com/library/en-us/cpref/html/frlrfsystemmarshalbyrefobjectclasstopic.asp

System.SerializableAttribute Class

Applied to any class, this attribute indicates that the class is serializable. For .NET Remoting, you need this each time you want to transport a class between the client and the server. Also often known as a marshal by value object, it is not accessed and executed on the remote server through an object reference; rather it is serialized, sent across the wire, and deserialized at the other endpoint. Therefore, objects that need to be sent as messages between a .NET Remoting client and a server have to have the SerializableAttribute applied.

Usage example:

```
[Serializable]
public class Customer
{
  public string Firstname;
  public string Lastname;
  public int Age;

  [NonSerialized]
  public int InternalNumber;

  public Customer(string first, string last, int age)
  {
    // ...
  }
}
```

Usually the .NET Framework serializer serializes all public and private members of a class. If you don't want specific members to be serialized, you can use the NonSerialized attribute for those members.

I use this attribute for all classes that are exchanged between the client and the server as messages. The first occurrence and explanation of this attribute can be found in Chapter 2 in the first .NET Remoting example.

More information on MSDN:
http://msdn.microsoft.com/library/en-us/cpref/html/frlrfsystemserializableattributeclasstopic.asp

System.Delegate Class

Delegates are safe references to either static or instance methods of classes. People programming in C++ could image a delegate as a safe version of function pointers. Delegates are used in many cases when programming .NET-based applications. The most common usage scenario for delegates is events. But in general, you can implement any function callback you need with delegates.

Because delegates are classes too, they can be used as function parameters as well as class members. For using delegates, you first have to define the structure of the delegate, which is made up of its name and the type of the return value, as well as the required parameters, including their types.

Usage example:

```
public delegate string MyDelegateName(int x, int y);

class MyDelegateTest
{
    [STAThread]
    static void Main(string[] args)
    {
    Console.WriteLine("Which delegate to use (1, 2): ");
    short sel = Int16.Parse(Console.ReadLine());
    if(sel == 1)
      DoActions(new MyDelegateName(MyFirstImplementation));
    else
      DoActions(new MyDelegateName(MySecondImplementation));
    Console.ReadLine();
    }

  static string MyFirstImplementation(int x, int y)
  {
    return (x+y).ToString();
  }

  static string MySecondImplementation(int x, int y)
  {
    return (x*y).ToString();
  }

  static void DoActions(MyDelegateName functionReference)
  {
    for(int i=0; i < 10; i++)
      for(int j=0; j < 2*i; j++)
        Console.WriteLine("-) {0}", functionReference(i, j));
  }
}
```

Delegates are used for the first time in Chapter 3 for implementing asynchronous calls. In Chapter 7, you can see delegates used for implementing events.

More information on MSDN:
`http://msdn.microsoft.com/library/en-us/cpref/html/frlrfsystemdelegateclasstopic.asp`

System.IAsyncResult Interface

The IAsyncResult interface represents the status of an asynchronous operation. It can be used in conjunction with the `BeginInvoke()` and `EndInvoke()` methods of a delegate. The `BeginInvoke()` method starts the asynchronous execution of the delegate and allows the application to do something different instead of waiting for the function to be finished. As soon as the application requires the results, it can call the delegate's `EndInvoke()` method, passing the IAsyncResult as a parameter. In this case, the application waits for the delegate to be finished and gets the actual return value.

Usage example:

```
IAsyncResult result = functionReference.BeginInvoke(i, j, null, null);
while(!result.IsCompleted)
{
  // wait for completion or do something else...
}
string ret = functionReference.EndInvoke(result);
```

I use the IAsyncResult interface for the first time in Chapter 3 and then in Chapter 7 for implementing an asynchronous method call in the examples.

More information on MSDN:
`http://msdn.microsoft.com/library/en-us/cpref/html/frlrfsystemiasyncresultclasstopic.asp`

System.Runtime.Remoting

Most of the time when writing .NET Remoting applications, you spend your time using classes defined in the root namespace System.Runtime.Remoting. The most important class in here is definitely the RemotingConfiguration class, which offers you the possibility of configuring .NET Remoting services as well as clients.

Basic Infrastructure Classes

The following classes are basic support classes that enable the .NET Remoting framework's core functionality.

ObjRef Class

Although not used directly, the ObjRef class is the secret behind accessing remote objects. It is the representation of a reference to an object running in a different application domain either

on the same or on a remote machine. Of course, object references can be passed between multiple instances so that any one of them can access the same object that is referenced through this class.

The ObjRef class is explained in detail in Chapter 3 when you encounter MarshalByRef as well as the capability of creating a multiserver configuration.

More information on MSDN:
`http://msdn.microsoft.com/library/en-us/cpref/html/frlrfsystemruntimeremotingobjrefclasstopic.asp`

ObjectHandle Class

Although I have not used this class directly in the examples, it is worth mentioning its existence. What the ObjRef class is for remote objects running in other application domains the ObjectHandle is for serialized classes passed between application domains. The object handle gives you the possibility of passing serialized types through an indirection between application domains.

This level of indirection can help you improve performance because as long as you don't call its `Unwrap` method, the metadata of the serialized type is not loaded into the application domain.

More information on MSDN:
`http://msdn.microsoft.com/library/en-us/cpref/html/frlrfsystemruntimeremotingobjecthandleclasstopic.asp`

RemotingConfiguration Class

The RemotingConfiguration class—as its name says—is the primary class for configuring .NET Remoting applications. You can use the class for registering your types either manually or through configuration files. This class cannot be used for registering channels; in that case, you have to use the ChannelServices class, which will be explained later in this appendix.

Usage examples:

```
// configure an object published on the server as a Singleton
RemotingConfiguration.RegisterWellKnownServiceType(
                            typeof(CustomerManager),
                            "CustomerManager.soap",
                            WellKnownObjectMode.Singleton);

// configure a remote object on the client residing under the following URL
RemotingConfiguration.RegisterWellKnownClientType(
                        typeof(IRemoteComponent),
                        "tcp://localhost:8080/MyRemote.rem");

// perform the configuration based on the contents of MyConfigFile.config
RemotingConfiguration.Configure("MyConfigFile.config");
```

You can see this class used for configuring .NET Remoting applications in almost every chapter in this book, starting with the first example in Chapter 2.

More information on MSDN:
`http://msdn.microsoft.com/library/en-us/cpref/html/frlrfsystemruntimeremotingremotingconfigurationclasstopic.asp`

RemotingServices Class

The RemotingServices class can be used for publishing remoted objects and proxies on a .NET Remoting server by calling static methods of this class. It can also be used to connect to a remote object by calling its `Connect` method, which basically does the same as the `Activator.GetObject` method.

Usage examples:

```
// publish an existing object on the server
MyClass cls = new MyClass(DateTime.Now);
RemotingServices.Marshal(cls,
          " MyRemote.rem",
          typeof(IRemoteComponent));

// create and use the proxy on the client
IRemoteComponent c = (IRemoteComponent)RemotingServices.Connect(
                                        typeof(IRemoteComponent),
                                        "tcp://localhost:8080/MyRemote.rem");
if(RemotingServices.IsObjectOutOfAppDomain(c)) {
  Console.WriteLine("What a surprise: object in another AppDomain!");
}
```

I use the RemotingServices class for the first time in Chapter 3 when publishing a created object. The intention is to enable passing parameters to the constructor, which is not possible when configuring Singleton or SingleCall objects through the methods offered by RemotingConfiguration (in which case objects are created automatically by the runtime).

More information on MSDN:
`http://msdn.microsoft.com/library/en-us/cpref/html/frlrfsystemruntimeremotingremotingservicesclasstopic.asp`

Configuration Classes

The following classes are used to manually configure the .NET Remoting functionality at runtime.

TypeEntry Class

The TypeEntry class is the general base class for various type configurations in the .NET Remoting runtime. Types that are configured for being available remotely or remote types used from within a client have to be configured by using the intended subclass of the TypeEntry class. Any type entries can be configured through either configuration files or the RemotingConfiguration class. The specific subclasses are explained in the chapters referenced in the discussions of these subclasses that follow.

In Chapter 4, I introduce the RemotingHelper class, which iterates through the configured entries in the current configuration. It uses the client-side subclasses of the TypeEntry class for doing so.

More information on MSDN:
`http://msdn.microsoft.com/library/en-us/cpref/html/frlrfsystemruntimeremotingtypeentryclasstopic.asp`

ActivatedServiceTypeEntry Class

The ActivatedServiceTypeEntry class is used for registering a class on a .NET Remoting service that is used as a client-activated object. In this case, object creation occurs when the client is sending an activation request message. Registration occurs either through configuration files or by calling the `RemotingConfiguration.RegisterActivatedServiceType()` method.

Usage example:

```
RemotingConfiguration.RegisterActivatedServiceType(
      new ActivatedServiceTypeEntry(typeof(IRemoteComponent)));

RemotingConfiguration.RegisterActivatedServiceType(typeof(IRemoteComponent));
```

Configuration example:

```
<configuration>
   <system.runtime.remoting>
      <application>
         <service>
            <activated type="MyNamespace.IRemoteComponent, MyAssemblyName" />
         </service>
      </application>
   </system.runtime.remoting>
</configuration>
```

I use the `RegisterActivatedServiceType()` method (and therefore configured an ActivatesServiceTypeEntry) for the first time in Chapter 3 when explaining the difference between server-activated and client-activated objects.

More information on MSDN:
`http://msdn.microsoft.com/library/en-us/cpref/html/frlrfsystemruntimeremotingactivatedservicetypeentryclasstopic.asp`

ActivatedClientTypeEntry Class

This class is the client's counterpart to the ActivatedServiceTypeEntry configuration on the server. It holds all the configuration information for a remote client-activated object on the client. After this method has been called, you can use either the `Activator.CreateInstance()` method or (when working with generated metadata and SoapSuds.exe) the `new` operator for creating a new instance of the remote object.

With the call to the `new` operator or the `Activator.CreateInstance()` method, the client sends a creation message to the server that leads the server to create a new instance with the parameters passed in the activation message. The server returns the object reference, and the runtime creates the TransparentProxy on the client.

Usage examples:

```
RemotingConfiguration.RegisterActivatedClientType(
          typeof(MyClass), "tcp://localhost:8080/MyRemote.rem");

RemotingConfiguration.RegisterActivatedClientType(
          new ActivatedClientTypeEntry(typeof(MyClass),
                          "tcp://localhost:8080/MyRemote.rem"));
```

Configuration example:

```
<configuration>
   <system.runtime.remoting>
      <application>
         <client url="http://localhost:8080" />
            <activated type="MyNamespace.MyClass, MyAssemblyName" />
         </client>
      </application>
   </system.runtime.remoting>
</configuration>
```

I use this method together with the corresponding configuration on the server for the first time in Chapter 3 when explaining the difference between SAO und CAO.

More information on MSDN:
`http://msdn.microsoft.com/library/en-us/cpref/html/frlrfsystemruntimeremotingactivatedclienttypeentryclasstopic.asp`

WellKnownServiceTypeEntry Class

The WellKnownServiceTypeEntry class allows you to publish server-activated objects in Singleton or SingleCall mode. In this case, the objects are created on the server and not through a creation message sent by the client, as is the case with activated types.

Usage examples:

```
RemotingConfiguration.RegisterWellKnownServiceType(
                              typeof(MyClass),
                              "MyRemoteObject.rem",
                              WellKnownObjectMode.Singleton);

RemotingConfiguration.RegisterWellKnownServiceType(
    new WellKnownServiceTypeEntry(typeof(MyClass),
                                  "MyRemoteObject.rem",
                                  WellKnownObjectMode.SingleCall));
```

Configuration example:

```
<configuration>
   <system.runtime.remoting>
      <application>
         <service>
            <wellknown type="MyNamespace.MyClass, MyAssembly"
                       objectUri="MyRemoteObject.rem"
                       mode="Singleton" />
         </service>
      </application>
   </system.runtime.remoting>
</configuration>
```

You can see server-activated objects used for the first time in Chapter 2 in the first .NET Remoting example. In Chapter 3, you can find details about the differences between server-activated and client-activated objects.

More information on MSDN:
http://msdn.microsoft.com/library/en-us/cpref/html/frlrfsystemruntimeremotingwellknownservicetypeentryclasstopic.asp

WellKnownClientTypeEntry Class

As with the ActivatedClientTypeEntry class, the WellKnownClientTypeEntry class holds the configuration of a server-activated object used by the client application. Therefore, it is the client's counterpart for the WellKnownServiceTypeEntry.

Usage examples:

```
RemotingConfiguration.RegisterWellKnownClientType(
                        typeof(IRemoteComponent),
                        "tcp://localhost:8080/MyRemoteObject.rem");

RemotingConfiguration.RegisterWellKnownClientType(
      new WellKnownClientTypeEntry(typeof(IRemoteComponent),
                                   "tcp://localhost:8080/MyRemoteObject.rem"));
```

Configuration example:

```
<configuration>
   <system.runtime.remoting>
      <application>
         <client>
            <wellknown
               type="MyNamespace.IRemoteComponent, MySharedAssembly"
               url="tcp://localhost:8080/RemoteType.rem" />
         </client>
      </application>
   </system.runtime.remoting>
</configuration>
```

In Chapters 2 and 3, I use this type of configuration for creating the first .NET Remoting example as well as explaining the difference between server-activated and client-activated objects.

More information on MSDN:
`http://msdn.microsoft.com/library/en-us/cpref/html/frlrfsystemruntimeremotingwellknownclienttypeentryclasstopic.asp`

WellKnownObjectMode Enumeration

This enumeration is used in conjunction with the RegisterWellKnownService type method for registering-server activated objects in .NET Remoting server components. It allows you to select the mode of the server-activated object, which can be Singleton or SingleCall.

I explain the difference between Singleton and SingleCall objects in Chapter 3 when presenting the details about server-activated objects.

More information on MSDN:
`http://msdn.microsoft.com/library/en-us/cpref/html/frlrfsystemruntimeremotingwellknownobjectmodeclasstopic.asp`

Exception Classes

The .NET Framework includes three main exceptions to convey additional information about the type of error.

RemotingException Class

This exception is thrown by the infrastructure when something has been going wrong when using .NET Remoting. The exception inherits from System.Exception and therefore offers you querying of the same types of information as a standard .NET exception.

More information on MSDN:
`http://msdn.microsoft.com/library/en-us/cpref/html/frlrfsystemruntimeremotingremotingexceptionclasstopic.asp`

RemotingTimeoutException Class

This exception inherits from the base class RemotingException and is thrown by the .NET Remoting infrastructure when the server (or the client in case of an event/callback) cannot be reached for a previously specified period of time. Timeouts can be specified through either the channel configuration or the RemotingClientProxy class, which is the base class for proxies generated via SoapSuds.exe.

More information on MSDN:
`http://msdn.microsoft.com/library/en-us/cpref/html/frlrfsystemruntimeremotingremotingtimeoutexceptionmemberstopic.asp`

ServerException Class

This exception is thrown by the infrastructure when the client connects to non–.NET-based components that are not able to throw exceptions on their own.

More information on MSDN:
http://msdn.microsoft.com/library/en-us/cpref/html/frlrfsystemruntimeremotingserverexceptionmemberstopic.asp

General Interfaces

The following interfaces are parts of the internal processing of remoting interactions.

IChannelInfo Interface

This interface provides a basic contract for carrying channel-specific data with an object reference (ObjRef) between the client and the server. This information is serialized with the ObjRef and transferred between the two actors so that the channel on the other side can read and leverage the information.

You can use this interface for extending the .NET Remoting infrastructure or reading channel-specific data in your custom code. In your own code, you can use the channels registered in your application domain to examine channel-specific data on the receiving side.

More information on MSDN:
http://msdn.microsoft.com/library/en-us/cpref/html/frlrfsystemruntimeremotingichannelinfoclasstopic.asp

IEnvoyInfo Interface

This interface allows you to pass context information used by message sinks between the client and the server. This information can be basically used for extending the .NET Remoting infrastructure.

More information on MSDN:
http://msdn.microsoft.com/library/en-us/cpref/html/frlrfsystemruntimeremotingiobjecthandleclasstopic.asp

IObjectHandle Interface

This is the base interface implemented by the ObjectHandle class referenced at the very beginning of this appendix. An ObjectHandle is a reference to serialized objects passed between application domains and allows you to avoid loading type information into an application domain where it is not needed. For more information, take a look at the "ObjectHandle Class" section of this appendix.

More information on MSDN:
http://msdn.microsoft.com/library/en-us/cpref/html/frlrfsystemruntimeremotingiobjecthandleclasstopic.asp

IRemotingTypeInfo Interface

When passing a reference of a remote object to a client, the ObjRef carries some type information with it. This type information can be queried through its `TypeInfo` property, which returns an object implementing this interface. You can do both querying the type name as well as asking for possible type casts through the interface's `CanCastTo` method.

More information on MSDN:
`http://msdn.microsoft.com/library/en-us/cpref/html/frlrfsystemruntimeremotingiremotingtypeinfoclasstopic.asp`

System.Runtime.Remoting.Channels

In this namespace, you can find the general interfaces as well as concrete implementations for .NET Remoting channels and message formatter classes. Channels are used for implementing the details of the transport protocol employed for calling remote components. The .NET Framework 1.0 and 1.1 ship with two prebuilt channels, one channel for TCP and another one for HTTP. With .NET 2.0, the infrastructure includes an IPC channel for interprocess communication.

General Interfaces and Classes

In this section, I have included the main interfaces and classes that are related to the system of extensible channels in the .NET Remoting framework.

IChannel Interface

This interface specifies the basic contract that has to be implemented by all channel objects. It specifies that each channel has to support at least two properties, `ChannelName` and `ChannelPriority`. The `ChannelName` property specifies a unique name for the channel, while the `ChannelPriority` property specifies which channel is given first chance to connect to a remote object. For server objects, it specifies the order in which channel data appears in the ObjRef passed to the client. Higher numbers indicate higher priority.

The IChannel interface also specifies that implementing classes must provide a `Parse` method. This method is used for parsing a complete URL and returning the URI for the current channel as well as the object's URI as an out parameter.

Usually you do not use the interface on its own, except if you want to write code that is independent of a specific channel implementation such as the HttpChannel. When implementing your own channel, you have to implement this interface.

Usage example:

```
// run through the configured channels
foreach(IChannel ch in ChannelServices.RegisteredChannels)
{
  Console.WriteLine("Channel: " + ch.ChannelName);
  if(ch is HttpClientChannel)
  {
    BaseChannelWithProperties chp = (BaseChannelWithProperties)ch;
    foreach(string key in chp.Properties.Keys)
    {
      Console.WriteLine("-) {0}={1}", key, chp.Properties[key]);
    }
  }
}
```

As this is the only interface that is useful for using from within your application directly, I will not explain the other interfaces of this namespace here, but in Appendix B.

More information on MSDN:
http://msdn.microsoft.com/library/en-us/cpref/html/frlrfsystemruntimeremotingchannelsichannelclasstopic.asp

ChannelServices Class

The ChannelServices class allows you to configure .NET Remoting channels for your application domain as well as enumerate registered channels and retrieve channel URIs. Each channel must have a unique name within an application domain. This class is also used by the RemotingConfiguration class when reading information out of configuration files.

Usage examples:

```
Hashtable props = new Hashtable();
props.Add("timeout", 20);
props.Add("proxyPort", "8080");
props.Add("proxyName", "myproxy");
props.Add("name", "My first channel");
props.Add("priority", 30);
props.Add("useDefaultCredentials", "true");

HttpClientChannel channel = new HttpClientChannel(props, null);
ChannelServices.RegisterChannel(channel);
```

Configuration example:

```
<configuration>
   <system.runtime.remoting>
      <application>
         <client>
            <wellknown type="RemoteType, RemoteAssembly"
                       url="http://localhost:8080/MyRemoteObject.rem" />
         </client>
         <channels>
            <channel ref="http" port="0" />
         </channels>
      </application>
   </system.runtime.remoting>
</configuration>
```

Channels are used in nearly all chapters throughout the book, either through configuration files or configuration in code. The first time you can see the use and explanation of channels is in Chapter 2, in the first .NET Remoting example. Configuration details can be found in Chapter 4.

According to configuration, you can either define channels within the <application> element or directly under <system.runtime.remoting>. When directly defining under <system.runtime.remoting>, channel templates, rather than channel instances, will be configured that can be referenced from within the <application> element using the ref attribute when defining the application's channel.

More information on MSDN:
http://msdn.microsoft.com/library/en-us/cpref/html/frlrfsystemruntimeremotingchannelschannelservicesclasstopic.asp

BinaryServerFormatterSinkProvider Class

This class provides the server with the functionality of serializing the message into binary format before sending it out through the channel and deserializing the incoming message before forwarding it to other channel sinks or the application. In terms of usage, you usually have to configure formatters if you want to configure one of the following properties for a sending channel:

- *includeVersions*: Specifies whether to include version information when sending serialized types across the wire.
- *strictBinding*: Specifies whether the exact version type is necessary for deserialization or not. If not, a version of the deserialized type must be installed in the GAC.
- *typeFilterLevel*: This is by far the most necessary attribute. Here you can specify which functionality will be permitted by the serializer during deserialization when receiving a serialized type. The property can be set to Low or Full, whereas Low restricts the types accepted by the deserializer (e.g., callbacks through delegates are not allowed with the Low setting). The default setting since .NET Framework 1.1 is Low.

Usage example:

```
BinaryServerFormatterSinkProvider sink1 = new BinaryServerFormatterSinkProvider();
sink1.TypeFilterLevel = TypeFilterLevel.Full;

SoapServerFormatterSinkProvider sink2 = new SoapServerFormatterSinkProvider();
sink2.TypeFilterLevel = TypeFilterLevel.Full;
sink2.Next = sink1;

HttpServerChannel channel = new HttpServerChannel("MyChannel", 8080, sink2);
ChannelServices.RegisterChannel(channel);
```

Configuration example:

```
<configuration>
  <system.runtime.remoting>
    <application>
      <channels>
        <channel ref="tcp" port="1234">
          <serverProviders>
            <formatter ref="binary"
                       typeFilterLevel="Full" />
            <formatter ref="soap"
                       typeFilterLevel="Full" />
          </serverProviders>
        </channel>
```

```
      </channels>
      <service>
        <wellknown type="Server.ServerImpl, Server"
                   objectUri="MyServer.rem"
                   mode="Singleton" />
      </service>
    </application>
  </system.runtime.remoting>
</configuration>
```

This class is used for the first time in Chapter 4 when the configuration options for the typeFilterLevel are explained.

More information on MSDN:
http://msdn.microsoft.com/library/en-us/cpref/html/frlrfsystemruntimeremotingchannelsbinaryserverformattersinkproviderclasstopic.asp

SoapServerFormatterSinkProvider Class

This class provides the server with the functionality of serializing the message into SOAP format before sending it out through the channel and deserializing the incoming message before forwarding it to other channel sinks or the application. In terms of usage, you usually have to configure formatters if you want to configure one of the settings described in the "BinaryServerFormatterSinkProvider Class" section earlier.

I use this class for the first time in Chapter 4 when explaining the typeFilterLevel attribute, which has changed from .NET Framework version 1.0 to 1.1 for security reasons. Also, take a look at the usage example in the "BinaryServerFormatterSinkProvider Class" section, where I also use the SoapServerFormatterSinkProvider class to configure the typeFilterLevel attribute.

More information on MSDN:
http://msdn.microsoft.com/library/en-us/cpref/html/frlrfsystemruntimeremotingchannelssoapserverformattersinkproviderclasstopic.asp

BinaryClientFormatterSinkProvider Class

This class is the opposite of the server's BinaryServerFormatterSinkProvider class. It is doing the deserialization and serialization of serializable types from and to binary format on the client side.

The typeFilterLevel attribute cannot be set on this class. If you want to configure this option for the client application, you have to use the BinaryServerFormatterSinkProvider as explained in Chapter 4 (within the serviceProviders section of the configuration files), or in the preceding two sections in this appendix.

More information on MSDN:
http://msdn.microsoft.com/library/en-us/cpref/html/frlrfsystemruntimeremotingchannelsbinaryclientformattersinkproviderclasstopic.asp

SoapClientFormatterSinkProvider Class

This class is the opposite of the server's BinaryServerFormatterSinkProvider class. It performs the deserialization and serialization of serializable types from and to SOAP format on the client side.

The typeFilterLevel attribute cannot be set on this class. If you want to configure this option for the client application, you have to use the SoapServerFormatterSinkProvider as explained in Chapter 4 (within the serviceProviders section of the configuration files), or in the two preceding sections in this appendix.

More information on MSDN:
`http://msdn.microsoft.com/library/en-us/cpref/html/frlrfsystemruntimeremotingchannelssoapclientformattersinkproviderclasstopic.asp`

BinaryClientFormatterSink Class

The BinaryClientFormatterSink class is a default formatter provided by the .NET Remoting framework for formatting messages in a binary format before they are sent across the wire to the remote server object. Internally this formatter sink leverages the binary formatters provided in the System.Runtime.Serialization.Formatters namespace.

As the sink is created by its corresponding provider, the BinaryClientFormatterSinkProvider, which is also used in configuration files or code for configuring the formatter's properties, you usually won't use this class directly in your own code.

More information on MSDN:
`http://msdn.microsoft.com/library/en-us/cpref/html/frlrfsystemruntimeremotingchannelsbinaryclientformattersinkclasstopic.asp`

BinaryServerFormatterSink Class

The BinaryServerFormatterSink class receives messages in binary format sent from the client and deserializes the binary stream back into its message format. Afterwards the message can be processed by the other sinks in the chain.

As the sink is created by its corresponding provider, the BinaryServerFormatterSinkProvider, which is also used in configuration files or code for configuring the formatter's properties, you usually won't use this class directly in your own code.

More information on MSDN:
`http://msdn.microsoft.com/library/en-us/cpref/html/frlrfsystemruntimeremotingchannelsbinaryserverformattersinkclasstopic.asp`

SoapClientFormatterSink Class

The SoapClientFormatterSink class is a default formatter provided by the .NET Remoting framework for formatting messages in SOAP-based format before they are sent across the wire to the remote server object. Internally this formatter sink leverages the SOAP formatters provided in the System.Runtime.Serialization.Formatters namespace.

As the sink is created by its corresponding provider, the SoapClientFormatterSinkProvider, which is also used in configuration files or code for configuring the formatter's properties, you usually won't use this class directly in your own code.

More information on MSDN:
`http://msdn.microsoft.com/library/en-us/cpref/html/frlrfsystemruntimeremotingchannelssoapclientformattersinkclasstopic.asp`

SoapServerFormatterSink Class

The SoapServerFormatterSink Class receives messages in SOAP format sent from the client and deserializes the SOAP structure back into its original message format. Afterwards the message can be processed by the other sinks in the chain.

As the sink is created by its corresponding provider, the SoapServerFormatterSinkProvider, which is also used in configuration files or code for configuring the formatter's properties, you usually won't use this class directly in your own code.

More information on MSDN:
`http://msdn.microsoft.com/library/en-us/cpref/html/frlrfsystemruntimeremotingchannelssoapserverformattersinkclasstopic.asp`

System.Runtime.Remoting.Channels.Http

This namespace includes client- and server-side channels for interactions via the HTTP protocol.

HttpChannel Class

This class provides an implementation of a channel using the HTTP protocol that is able to send as well as receive messages across the wire. Actually, it combines HttpClientChannel and the HttpServerChannel in one implementation. By default, this channel uses the SOAP formatter classes to transmit messages in SOAP format across the wire.

Usage example:

```
Hashtable props = new Hashtable();
props.Add("name", "My first channel");
props.Add("priority", 30);

HttpChannel channel = new HttpChannel(props, null, null);
ChannelServices.RegisterChannel(channel);
```

Configuration example:

```
<configuration>
   <system.runtime.remoting>
      <application>
         <client>
            <wellknown type="MyNamespace.MyClass, MySharedAssembly"
                       url="http://localhost:8080/MyRemoteObject.rem" />
```

```
        </client>
        <channels>
           <channel ref="http" port="0" />
        </channels>
      </application>
   </system.runtime.remoting>
</configuration>
```

On the client side, a port has to be configured when the client needs to receive callbacks from the server. In this case, you can also use the <serviceProviders> element to configure server channels with typeFilterLevel.

I use this channel across multiple examples throughout the whole book. Details about the channel and its configuration can be found in Chapter 4.

More information on MSDN:
http://msdn.microsoft.com/library/en-us/cpref/html/frlrfsystemruntimeremotingchannelshttphttpchannelclasstopic.asp

HttpClientChannel Class

While the HttpChannel class implements both the sending and receiving part of a channel, the HttpClientChannel class only implements the client-side part of the channel. By default, all messages are passed through the SOAP formatter, which means that messages are transmitted via SOAP/XML to the server.

Usage example:

```
HttpClientChannel channel = new HttpClientChannel("My Client Channel", null);
ChannelServices.RegisterChannel(channel);
```

Configuration example:

```
<configuration>
<system.runtime.remoting>
   <channelSinkProviders>
      <channel type="System.Runtime.Remoting.Channels.Http.HttpChannel,
                    System.Runtime.Remoting" id="httpbinary" >
         <clientProviders>
            <formatter type="System.Runtime.Remoting.Channels.
                             BinaryClientFormatterSinkProvider,
                             System.Runtime.Remoting" />
         </clientProviders>
      </channel>
   </channels>
   <application>
      <channels>
         <channel ref="httpbinary" />
      </channels>
      <client>
```

```
        <wellknown url="http://localhost:80/MyRemoteObject.rem"
                   type="MyNamespace.MyRemoteObject, SharedAssembly" />
      </client>
   </application>
</system.runtime.remoting>
</configuration>
```

The preceding configuration example demonstrates how the HTTP channel can be used with the binary formatter on the client side. On the server side, the same configuration would be used with the <serverProviders> element instead of the <clientProviders> element.

More information on MSDN:
http://msdn.microsoft.com/library/en-us/cpref/html/frlrfsystemruntimeremotingchannelshttphttpclientchannelclasstopic.asp

HttpServerChannel Class

While the HttpChannel class implements both the client and server part, this channel only implements the server-side part of the channel. By default, the channel accepts messages in both SOAP and binary format.

Configuration and usage is very similar to the HttpClientChannel. In configuration files, you use the <serverProviders> element instead of <clientProviders>.

More information on MSDN:
http://msdn.microsoft.com/library/en-us/cpref/html/frlrfsystemruntimeremotingchannelshttphttpserverchannelclasstopic.asp

System.Runtime.Remoting.Channels.Tcp

This namespace includes client- and server-side channels for interactions via a proprietary TCP-based protocol.

TcpChannel Class

As with the HttpChannel class, the TcpChannel class implements both the client- and the server-channel part for transmitting messages across the wire using the TCP protocol. Therefore, it is a combination of the TcpClientChannel as well as the TcpServerChannel. By default, it accepts and transmits messages in binary format.

Usage example:

```
TcpChannel channel = new TcpChannel(4711);
ChannelServices.RegisterChannel(channel);
```

Configuration example:

```
<configuration>
  <system.runtime.remoting>
    <application>
      <channels>
```

```
        <channel ref="tcp" port="1234">
          <serverProviders>
            <formatter ref="binary" />
          </serverProviders>
        </channel>
      </channels>
      <service>
        <wellknown type="Server.ServerImpl, Server"
                   objectUri="MyServer.rem" mode="Singleton" />
      </service>
    </application>
  </system.runtime.remoting>
</configuration>
```

For details about the TcpChannel class and its configuration options, take a closer look at the descriptions in Chapter 4.

More information on MSDN:
http://msdn.microsoft.com/library/en-us/cpref/html/frlrfsystemruntimeremotingchannelstcptcpchannelclasstopic.asp

TcpClientChannel Class

While the TcpChannel class implements both the client- and the server-side part of the channel, this implementation can be used on clients only. In configuration files, it is used together with the <clientProviders> part for configuration of the message sinks and formatters of the channel.

Usage example:

```
TcpClientChannel channel = new TcpClientChannel("My Tcp Channel", null);
ChannelServices.RegisterChannel(channel);
```

Configuration example:

```
<configuration>
  <system.runtime.remoting>
    <application name="FirstServer">
      <channels>
        <channel ref="tcp" />
      </channels>
      <client>
        <wellknown type="General.IRemoteFactory, General"
                   url="tcp://localhost:1234/MyServer.rem" />
      </client>
    </application>
  </system.runtime.remoting>
</configuration>
```

Configuration is similar to HttpClientChannel as well as TcpChannel in general. Therefore, for details, look at the corresponding sections in this appendix. For further details about channel configuration, take a closer look at Chapter 4.

More information on MSDN:
`http://msdn.microsoft.com/library/en-us/cpref/html/frlrfsystemruntimeremotingchannelstcptcpclientchannelclasstopic.asp`

TcpServerChannel Class

Whereas the TcpClientChannel class implements the client-side channel, this class implements the server-side channel only. By default, it accepts messages in either SOAP or binary format. Configuration and usage is very similar to TcpChannel (and TcpClientChannel) except that you use the <serverProviders> element in the configuration files when configuring formatters and message sinks for this channel.

More information on MSDN:
`http://msdn.microsoft.com/library/en-us/cpref/html/frlrfsystemruntimeremotingchannelstcptcpserverchannelclasstopic.asp`

System.Runtime.Remoting.Lifetime

This namespace includes classes that are used to implement and customize the lease-based lifetime management system.

ILease Interface

The ILease interface defines the lifetime lease object that is used by .NET Remoting LifetimeServices. The ILease interface allows you to define the properties that will be used by LifetimeServices for evaluating how long a server-side object will be kept alive.

It enables the server object to define several types of timeouts. Based on these timeout values, the lifetime service decides when the server-side object can be destroyed. For configuring the lifetime of a server object, you have to override the `InitializeLifetimeService()` method of the MarshalByRefObject class.

Usage example:

```
public override object InitializeLifetimeService()
{
   Console.WriteLine("MyRemoteObject.InitializeLifetimeService() called");
   ILease lease = (ILease)base.InitializeLifetimeService();
   if (lease.CurrentState == LeaseState.Initial)
   {
      lease.InitialLeaseTime = TimeSpan.FromMilliseconds(10);
      lease.SponsorshipTimeout = TimeSpan.FromMilliseconds(10);
      lease.RenewOnCallTime = TimeSpan.FromMilliseconds(10);
   }
   return lease;
}
```

Configuration example:

```
<configuration>
   <system.runtime.remoting>
      <application>
         <lifetime
            leaseTimeout="10M"
            renewOnCallTime="5M"
            leaseManagerPollTime="30S"
         />
      </application>
   </system.runtime.remoting>
</configuration>
```

I use the ILease interface the first time in Chapter 3 in the section "Managing Lifetime." More details about lifetime management can be found at the very beginning of Chapter 7. Details about configuring lease times can be found in Chapter 4.

More information on MSDN:
http://msdn.microsoft.com/library/en-us/cpref/html/frlrfsystemruntimeremotinglifetimeileaseclasstopic.asp

ISponsor Interface

Every object that wants to request a lease renewal for a server-side object must implement the ISponsor interface. By implementing this interface, the object can become a sponsor by registering itself with the lease manager. A sponsor can reside on the client or on the server or on any other machine too. The only requirement is that it is reachable by the .NET Remoting infrastructure.

If the sponsor is used on the client, the client becomes a server itself as the .NET Remoting infrastructure on the server-side object tries to contact the sponsor for asking whether or not the TTL of the server object should be renewed after the lease time has expired.

Usage example:

```
// creation of a sponsor class
public class MySponsor: MarshalByRefObject, ISponsor
{

   public bool doRenewal = true;

   public TimeSpan Renewal(System.Runtime.Remoting.Lifetime.ILease lease)
   {
      Console.WriteLine("{0} SPONSOR: Renewal() called", DateTime.Now);

      if (doRenewal)
      {
         Console.WriteLine("{0} SPONSOR: Will renew (10 secs)", DateTime.Now);
         return TimeSpan.FromSeconds(10);
      }
```

```
      else
      {
         Console.WriteLine("{0} SPONSOR: Won't renew further", DateTime.Now);
         return TimeSpan.Zero;
      }
   }
}

public class MyApplication
{
  public static void Main(string[] args)
  {
    String filename = "client.exe.config";
    RemotingConfiguration.Configure(filename);

    SomeCAO cao = new SomeCAO();
    ILease le = (ILease) cao.GetLifetimeService();
    MySponsor sponsor = new MySponsor();
    le.Register(sponsor);

    // do something here ...

    // unregister the lease at the end
    le.Unregister(sponsor);
  }
}
```

You can see the ISponsor interface used for the first time in Chapter 7 in the "Managing Lifetime" section.

More information on MSDN:
http://msdn.microsoft.com/library/en-us/cpref/html/frlrfsystemruntimeremotinglifetimeisponsorclasstopic.asp

ClientSponsor Class

The ClientSponsor class provides a default implementation for a client-side sponsor. It allows you to set the `RenewalTime` property from outside and can be registered with the lifetime services as sponsor.

I have not used the default implementation in this book, but more information on implementing the ISponsor interface on your own can be found in Chapter 7.

More information on MSDN:
http://msdn.microsoft.com/library/en-us/cpref/html/frlrfsystemruntimeremotinglifetimeclientsponsorclasstopic.asp

LifetimeServices Class

This class enables you to control the lifetime infrastructure of .NET Remoting. Therefore, it gives you access to an ILease object for configuring several timeout values or registering sponsors.

Usage examples:

```
// LifetimeServices and client-activated objects
ILease le = (ILease) cao.GetLifetimeService();
MySponsor sponsor = new MySponsor();
le.Register(sponsor);

// LifetimeServices and server-activated objects
class ServerExampleClass: MarshalByRefObject
{
   public override object InitializeLifetimeService()
   {
      ILease tmp = (ILease) base.InitializeLifetimeService();
      if (tmp.CurrentState == LeaseState.Initial)
      {
         tmp.InitialLeaseTime = TimeSpan.FromSeconds(5);
         tmp.RenewOnCallTime = TimeSpan.FromSeconds(1);
      }
      return tmp;
   }
}
```

The LifetimeServices is used for the first time in Chapter 3 when I discuss the fundamentals about lifetime. More details can be found in Chapter 7.

More information on MSDN:
`http://msdn.microsoft.com/library/en-us/cpref/html/frlrfsystemruntimeremotinglifetimelifetimeservicesclasstopic.asp`

LeaseState Enumeration

The ILease interface has one property called `CurrentState`. This property enables the developer to query the current lease state of a server object. This state can be one of the following:

- *Activate*: Lease is activated and not expired
- *Expired*: Lease has expired and cannot be renewed
- *Initial*: Lease has been created but not yet activated
- *Renewing*: Lease has been expired and is seeking sponsorship
- *Null*: Lease is not initialized

More information on MSDN:
`http://msdn.microsoft.com/library/en-us/cpref/html/frlrfsystemruntimeremotinglifetimeleasestateclasstopic.asp`

System.Runtime.Remoting.Messaging

In this namespace, you will find classes that are used for transmitting and controlling transmitted messages between remoting clients and servers. Basically, the .NET Remoting infrastructure uses messages for communication between the client and the server. These messages are either serialized binary or in SOAP format.

Within the messaging namespace are classes that allow you to send some additional information in the message header (context) as well as control the way messages are sent and responses are evaluated in .NET Remoting applications.

AsyncResult Class

The AsyncResult class provides a default implementation of the IAsyncResult interface introduced in Chapters 3 and 7 in the sections "Asynchronous Calls" and "Remoting Events." I always use the IAsyncResult interface in this book's examples, as I had no specific reason for using this class. The class provides two additional methods. They allow you to complete the method call explicitly through the `Complete()` method and let you verify whether `EndInvoke()` has completed successfully.

If you require one of these two methods, you can use the AsyncResult class instead of the interface. If not, there is no reason for not using just the interface as done in the examples in the book.

More information on MSDN:
`http://msdn.microsoft.com/library/en-us/cpref/html/frlrfsystemruntimeremotingmessagingasyncresultclasstopic.asp`

CallContext Class

The CallContext class allows you to carry additional properties within the message exchanged between the client and the server. Therefore, you can include some additional metadata that can be used either by the server or by the client. This metadata has nothing to do with the actual business logic but more with some infrastructural topics.

Typical examples are things like security-related information as authentication method and encryption method. Another example is including some message IDs that can be used for implementing something like long-running business transactions in your system.

Usage examples:

```
// define a context object in the shared assembly (must be serializable!)
[Serializable]
public class MyContextObject : ILogicalThreadAffinative
{
  public DateTime RequestTime;
  public Guid ServerResponseGuid;
}

// example for using the context object in the client application
public static void Main(string[] args)
```

```
{
  HttpChannel channel = new HttpChannel();
  ChannelServices.RegisterChannel(channel);

  ICustomerManager mgr = (ICustomerManager) Activator.GetObject(
                              typeof(ICustomerManager),
                              "http://localhost:1234/CustomerManager.soap");
  Console.WriteLine("Client.Main(): Reference to CustomerManager acquired");

  MyContextObject ctx = new MyContextObject();
  ctx.RequestTime = DateTime.Now;
  ctx.ServerResponseGuid = Guid.Empty;
  CallContext.SetData("MyContext", ctx);

  Customer cust = mgr.GetCustomer(4711);

  MyContextObject ret = (MyContextObject)CallContext.GetData("MyContext");
  Console.WriteLine("Metadata: {0}", ret.ServerResponseGuid.ToString());
}

// example for using the context object in the server application
class CustomerManager: MarshalByRefObject, ICustomerManager
{
  public Customer GetCustomer(int id)
  {
    Console.WriteLine("CustomerManager.getCustomer): Called");
    Customer tmp = new Customer();
    tmp.FirstName = "John";
    tmp.LastName = "Doe";
    tmp.DateOfBirth = new DateTime(1970,7,4);

    object ctxData = CallContext.GetData("MyContext");
    if(ctxData != null)
    {
      MyContextObject ctx = (MyContextObject)ctxData;
      Console.WriteLine("Request sent at {0}",
                        ctx.RequestTime.ToString("dd.MM.YYYY - hh.mm.ss.SS"));
      ctx.ServerResponseGuid = Guid.NewGuid();
    }
    return tmp;
  }
}
```

I use CallContext for the first time in Chapter 9 when talking about contexts.

More information on MSDN:
http://msdn.microsoft.com/library/en-us/cpref/html/
frlrfsystemruntimeremotingmessagingcallcontextclasstopic.asp

LogicalCallContext Class

The LogicalCallContext class is a special version of the CallContext class that is used during method calls to remote objects. The CallContext class itself is used for sharing data across method calls in a single logical thread but not across logical threads or across application domains. As soon as it comes to communication with other threads or remote application domains, the CallContext class automatically creates a LogicalCallContext, which is used for transmitting data to the other application domain.

This is done only if transmitted context objects implement the ILogicalThreadAffinative interface. In any other case, the context objects are kept for the logical thread only, and are not transmitted to the other application domain.

You don't use this object directly, as the CallContext object automatically creates the LogicalCallContext when transmitting context information to the other application domain.

More information on MSDN:
`http://msdn.microsoft.com/library/en-us/cpref/html/frlrfsystemruntimeremotingmessaginglogicalcallcontextclasstopic.asp`

OneWayAttribute Class

This special attribute allows you to mark a method of a .NET Remoting server as a one-way method. One-way methods do not have any return values, out values, or ref parameters. In the case of remoting, the call message is sent by client objects without verifying any return values or success on calling the remote method.

Usage example:

```
public class BroadcastEventWrapper: MarshalByRefObject {
   // ...some other class members ...
   [OneWay]
   public void SampleOneWay (String msg) {
      // Do something here without returning anything
   }
}
```

Details about the OneWayAttribute class and its advantages and disadvantages can be found in Chapter 7.

More information on MSDN:
`http://msdn.microsoft.com/library/en-us/cpref/html/frlrfsystemruntimeremotingmessagingonewayattributeclasstopic.asp`

System.Runtime.Remoting.Metadata

The System.Runtime.Remoting.Metadata namespace includes classes and attributes for controlling the processing and serialization of objects and fields when using SOAP as your message format. These classes can be used for specifying XML element and attribute names for serialization as well as controlling the SOAP header itself.

The attributes explained in this section and introduced in this namespace provide similar functionality to the attributes introduced with the XML serialization (System.Xml.Serialization), which is used for XML and Web Services.

The attributes introduced in this section are used for the first time in the book in Chapter 8.

SoapAttribute Class

The SoapAttribute class is not used directly on your serializable types. It provides the base functionality for all attributes explained in this section.

More information on MSDN:
http://msdn.microsoft.com/library/en-us/cpref/html/frlrfsystemruntimeremotingmetadatasoapattributeclasstopic.asp

SoapTypeAttribute Class

The SoapTypeAttribute class is the most important of all the attributes explained in this section. It can be applied to classes and structures only, and specifies general attributes of the type like the XML root element name and, of course, the namespace that should be used for serialization.

Usage example:

```
[Serializable()]
[SoapTypeAttribute(XmlNamespace="MyXmlNamespace")]
public class TestSimpleObject
{
    public int member1;

    [SoapFieldAttribute(XmlElementName="MyXmlElement")]
    public string member2;

    // a field that is not serialized
    [NonSerialized()]
    public string member3;
}
```

More information on MSDN:
http://msdn.microsoft.com/library/en-us/cpref/html/frlrfsystemruntimeremotingmetadatasoaptypeattributeclasstopic.asp

SoapFieldAttribute Class

This attribute can be applied on a field of a serializable type and controls serialization of this field. Serialization is performed by the SOAP formatter that evaluates this attribute and then serializes the field according to the information in this attribute. If not present, it assumes some defaults like taking the fieldname for the name of the XML element serialized into the SOAP message.

More information on MSDN:
http://msdn.microsoft.com/library/en-us/cpref/html/frlrfsystemruntimeremotingmetadatasoapfieldattributeclasstopic.asp

SoapMethodAttribute Class

Other than the attributes introduced until now in this section, this attribute is not applied to a serializable type but on a method that can be invoked remotely. The SoapSuds.exe tool uses these attributes for the generation of appropriate client proxy classes. It controls the SOAPAction that will be serialized into the SOAP header.

More information on MSDN:
`http://msdn.microsoft.com/library/en-us/cpref/html/frlrfsystemruntimeremotingmetadatasoapmethodattributeclasstopic.asp`

SoapParameterAttribute Class

This attribute can be used together with SoapMethodAttribute. While SoapMethodAttribute is applied on the method itself, this attribute is applied on the method's parameters and dictates how parameters are serialized into the SOAP message (e.g., XML element names for parameters).

More information on MSDN:
`http://msdn.microsoft.com/library/en-us/cpref/html/frlrfsystemruntimeremotingmetadatasoapparameterattributeclasstopic.asp`

SoapOption Enumeration

The SoapTypeAttribute class includes a property, `SoapOptions`, that can be set to one of the values defined in this enumeration. This attribute controls how type information is included in the serialized SOAP message.

More information on MSDN:
`http://msdn.microsoft.com/library/en-us/cpref/html/frlrfsystemruntimeremotingmetadatasoapattributeclasstopic.asp`

System.Runtime.Remoting.Services

Within the System.Runtime.Remoting.Services namespace you will find several classes that provide additional remoting services to the .NET Framework. Here you can find a class for interoperating with Enterprise Services or the base class for proxies created via SoapSuds.exe.

By far the most important class is the TrackingServices class, which allows you to plug your own components into the marshaling, unmarshaling, and disconnection processes of remote objects.

EnterpriseServicesHelper Class

The EnterpriseServicesHelper class provides some APIs for communicating with unmanaged classes outside your own application domain.

More information on MSDN:
`http://msdn.microsoft.com/library/en-us/cpref/html/frlrfsystemruntimeremotingservicesenterpriseserviceshelperclasstopic.asp`

RemotingClientProxy Class

This class is the abstract base class for any client-side proxy for well-known objects generated by the SoapSuds.exe utility. It defines a set of frequently used properties for SoapSuds-generated proxies.

More information on MSDN:
`http://msdn.microsoft.com/library/en-us/cpref/html/frlrfsystemruntimeremotingservicesremotingclientproxyclasstopic.asp`

ITrackingHandler Interface

The ITrackingHandler interface defines functionality that allows an object to be notified when the .NET Remoting infrastructure marshals, unmarshals, or disconnects an object from its proxy.

Interface definition:

```
public interface ITrackingHandler
{
 void DisconnectedObject(object obj);
 void MarshaledObject(object obj, ObjRef or);
 void UnmarshaledObject(object obj, ObjRef or);
}
```

The preceding interface defines three methods that have to be implemented by your own tracking handler class. This implementation has to be registered with the TrackingServices class, which is defined in the same namespace.

TrackingHandlers can be used when you need to be notified for one or more of these events. In this case, you can implement functionality such as deterministic resource management or object pooling for remoting objects.

For example, if an object disconnects from its proxy, you can free unmanaged resources by catching the DisconnectObject event and calling the `Dispose()` method of your wrapper classes for the unmanaged resources to free them at once. You can also restore any resources when an object gets marshaled again if you need them for subsequent processing immediately.

More information on MSDN:
`http://msdn.microsoft.com/library/en-us/cpref/html/frlrfsystemruntimeremotingservicesitrackinghandlerclasstopic.asp`

TrackingServices Class

The TrackingServices class provides the basic infrastructure for registering your own tracking handlers. This enables you to catch the events explained in the description of the ITrackingHandler interface.

Tracking those events enables creation of mechanisms like object pooling or improved resource management in terms of releasing resources on object disconnection and restoring them when a new reference has been acquired (marshaling and unmarshaling).

Usage example:

```
// Part I:
// create your own tracking handler
public class SampleTrackingHandler : ITrackingHandler
{
   public void MarshaledObject(object obj, ObjRef or)
   {
      // write logging information
      // restore other objects from a pool on the server side
   }

   public void UnmarshaledObject(object obj, ObjRef or)
   {
      // write logging information or
   }

   public void DisconnectedObject(object obj)
   {
     // write logging code or
     // release any resources not required when disconnected
   }
}

// Part II:
// register your tracking handler
TrackingServices.RegisterTrackingHandler(objHandler) ;
// ...
// perform your operations here...
// ...
// unregister your handler if catching events not necessary anymore
TrackingServices.UnregisterTrackingHandler(objHandler) ;
```

More information on MSDN:
http://msdn.microsoft.com/library/en-us/cpref/html/
frlrfsystemruntimeremotingservicestrackingservicesclasstopic.asp

System.Runtime.Serialization

This namespace contains all types that are used by the generic .NET serializer for serializing and deserializing any types of objects. I have demonstrated some of the classes, interfaces, and enumerations for changing typeFilterLevel attribute on formatter sinks, as well as specifically in the versioning chapter, for controlling the serialization and deserialization of types transmitted between the client and the server.

As the generic serializer could fill a book on its own, I will only explain the parts that I use in the samples in this book.

ISerializable Interface

Usually serialization is done by the .NET serialization runtime and formatters automatically after an object has been marked with the [Serializable] attribute. In this case, the serializer implements a default behavior for serializing and deserializing objects. If an object wants to override this default behavior, it has to implement the ISerializable interface.

If an object implements ISerializable, the serialization runtime calls the interface's `GetObjectData()` method for getting the serialized version of the object. When deserializing the runtime, look for a special constructor in the class. Both the `GetObjectData()` method as well as the special constructor get two objects as parameters—a SerializationInfo class as well as a StreamingContext class, which are explained later in this section.

Usage example:

```
[Serializable]
public class Customer: ISerializable {
   public String FirstName;
   public String LastName;
   public DateTime DateOfBirth;
   public String Title;

   public Customer (SerializationInfo info, StreamingContext context) {
      FirstName = info.GetString("FirstName");
      LastName = info.GetString("LastName");
      DateOfBirth = info.GetDateTime("DateOfBirth");
      try {
         Title = info.GetString("Title");
      } catch (Exception e) {
         Title = "n/a";
      }
   }

   public void GetObjectData(SerializationInfo info,      StreamingContext context)
   {
      info.AddValue("FirstName",FirstName);
      info.AddValue("LastName",LastName);
      info.AddValue("DateOfBirth",DateOfBirth);
      info.AddValue("Title",Title);
   }
}
```

The preceding code is the sample code introduced in Chapter 8 for getting a serializable type that continues working with older versions of its own. I also introduced the usage of the interface for implementing advanced versioning concepts in Chapter 8 where the serializable type doesn't lose information when working with newer and older versions of its own.

More information on MSDN:
`http://msdn.microsoft.com/library/en-us/cpref/html/frlrfsystemruntimeserializationiserializableclasstopic.asp`

SerializationInfo Class

The SerializationInfo class holds the data that is serialized or that should be deserialized during the serialization process. In the preceding code example, SerializationInfo is used for writing the customer's data during the serialization process and reading the data during deserialization.

Usage example:

```
   public void GetObjectData(SerializationInfo info, StreamingContext context)
   {
      info.AddValue("FirstName",FirstName);
      info.AddValue("LastName",LastName);
      info.AddValue("DateOfBirth",DateOfBirth);
      info.AddValue("Title",Title);
   }
}
```

You can see this class in Chapter 8, where it is used for implementing custom serialization logic for versioning of the serialized types. An important fact is that you can also add subobjects to SerializationInfo as long as they are serializable on their own. But also you have to know up front what to serialize and what needs to be deserialized. The SerializationInfo doesn't include functionality for iterating through all serialized members during deserialization. Therefore, if you want to dynamically add different data to SerializationInfo, you have to work with objects like an ArrayList.

More information on MSDN:
`http://msdn.microsoft.com/library/en-us/cpref/html/frlrfsystemruntimeserializationserializationinfoclasstopic.asp`

StreamingContext Structure

The second parameter of the `GetObjectData()` method and the special constructor used for custom serialization logic is StreamingContext. This structure gives you some additional information about the serialization currently done like the source and the purpose for the serialization. The purpose can be queried through the class's `Context` property, which is of the type StreamingContextStates.

The StreamingContextStates enumeration gives you information about the source of serialization, for example, object cloning (Clone), file persistence (Persistence), and, of course, .NET Remoting.

More information on MSDN:
`http://msdn.microsoft.com/library/en-us/cpref/html/frlrfsystemruntimeserializationstreamingcontextclasstopic.asp`

More information about the StreamingContextStates enumeration on MSDN:
`http://msdn.microsoft.com/library/en-us/cpref/html/frlrfsystemruntimeserializationstreamingcontextstatesclasstopic.asp`

SerializationException Class

This exception is thrown when an error occurs during serialization or deserialization. Causes for a serialization exception might be trying deserialize a stream that does contain wrong or incomplete data or data for the wrong version of the serialized type.

More information on MSDN:
`http://msdn.microsoft.com/library/en-us/cpref/html/frlrfsystemruntimeserializationserializationexceptionclasstopic.asp`

System.Runtime.Serialization.Formatters

This namespace includes the necessary infrastructure for runtime formatting, which is internally used by the built-in formatters for .NET Remoting.

SoapFault Class

The SoapFault class represents an error that occurred when calling a remote method using SOAP. This class is used for carrying error and status information within the SOAP message. Although I have not used the class within the book, it should be mentioned here because SOAP faults are the standard for transferring error information when calling remote services via SOAP. Therefore, they are important when it comes to Web Services, too.

More information on MSDN:
`http://msdn.microsoft.com/library/en-us/cpref/html/frlrfsystemruntimeserializationformatterssoapfaultclasstopic.asp`

SoapMessage Class

The SoapMessage class is a representation of the metadata transferred within the SOAP message when calling a remote service/method via SOAP. It holds the method name as well as the names and types of the parameters required during serialization and deserialization of SOAP RPC messages. The corresponding counterpart for Web Services (which are usually not SOAP RPC) can be found in the System.Web.Services.Protocols namespace.

More information on MSDN:
`http://msdn.microsoft.com/library/en-us/cpref/html/frlrfsystemruntimeserializationformatterssoapmessageclasstopic.asp`

TypeFilterLevel Enumeration

The TypeFilterLevel enumeration was introduced with .NET Framework 1.1 for security reasons in the deserialization process. Theoretically, an attacker could leverage the process of deserialization. To avoid this possible security problem, TypeFilterLevel has been added to the serialization infrastructure.

With a TypeLevelFilter set to low, not all types will be deserialized by the serialization runtime. If some types such as delegate types are included in the messages transmitted between the client and the server, a SerializationException will be thrown.

Usage example:

```
// configure the formatters for the channel
BinaryServerFormatterSinkProvider formatterBin =
                  new BinaryServerFormatterSinkProvider();
formatterBin.TypeFilterLevel = TypeFilterLevel.Full;

// register the channels
IDictionary dict = new Hashtable();
dict.Add("port", "1234");

TcpChannel channel = new TcpChannel(dict, null, formatterBin);
ChannelServices.RegisterChannel(channel);

// register the wellknown service
RemotingConfiguration.RegisterWellKnownServiceType(typeof(ServerImpl),
```

Configuration example:

```
<configuration>
  <system.runtime.remoting>
    <application>
      <channels>
        <channel ref="tcp" port="1234">
          <serverProviders>
            <formatter ref="binary"
                       typeFilterLevel="Low" />
          </serverProviders>
        </channel>
      </channels>
      <service>
        <wellknown type="Server.ServerImpl, Server"
                   objectUri="MyServer.rem"
                   mode="Singleton" />
      </service>
    </application>
  </system.runtime.remoting>
</configuration>
```

I introduce TypeFilterLevel in Chapter 4 in the discussion on configuring channels. In configuration files, TypeFilterLevel is configured through the formatter sink providers within the <serverProviders> element of the corresponding channels.

More information on MSDN:
`http://msdn.microsoft.com/library/en-us/cpref/html/frlrfsystemruntimeserializationformatterstypefilterlevelclasstopic.asp`

Summary

In this appendix, you've seen an overview of the most important classes you'll encounter when using .NET Remoting. In the next appendix, this overview will be extended to cover the namespaces, classes, and interfaces that are used to extend the remoting framework.

APPENDIX B

.NET Remoting Extensibility Reference

In most cases, extending the .NET Remoting infrastructure means creating classes that implement specific interfaces of the framework. Within this appendix, you'll find a brief description of the interfaces needed for extending the .NET Remoting framework with your own channels and formatters, as well as message sinks. Some of the interfaces you'll find in this appendix might have been described in the previous appendix, too, but from within another context.

The namespaces where you can find the interfaces described in this appendix are as follows:

- System.Runtime.Remoting.Messaging
- System.Runtime.Remoting.Activation
- System.Runtime.Remoting.Proxies
- System.Runtime.Remoting.Channels

You can find details about the interfaces and how to use them in the chapters of the second part of the book.

System.Runtime.Remoting.Messaging

Basically, the .NET Remoting infrastructure uses messages for communicating with remote objects. Messages are used not only for calling remote objects, but also for activating them through so-called activation messages. A message carries all the information that is necessary for the remote object for appropriate processing. This means it consists of metadata like action identifiers as well as the actual user data. You can find all the interfaces for the basic messaging infrastructure of the framework in this namespace.

IMessage Interface

Each communication with remote objects is based on messages that are sent across the wire. The IMessage interface is the base interface for messages and therefore contains communication data sent between two parties.

Basically, the IMessage interface defines a dictionary object containing the message properties only. Within one .NET Remoting object, the message is passed through a set of sinks

to the transport channel, which is responsible for transmitting the message across the wire to the remote object. On the remote side, the object is passed through the receiving transport channel as well as a set of sinks to the actual object processing the message.

Interface definition:

```
public interface IMessage
{
 IDictionary Properties { get; }
}
```

Basically, implementations or subtypes of this interface add additional properties that allow easier access to the contents of the message as can be seen with the MethodCall class implementation later in this chapter.

Chapter references:

- Chapter 11: Inside the Framework
- Chapter 12: Creation of Sinks
- Chapter 13: Extending .NET Remoting
- Chapter 14: Developing a Transport Channel
- Chapter 15: Context Matters

More information on MSDN:
`http://msdn.microsoft.com/library/en-us/cpref/html/frlrfsystemruntimeremotingmessagingimessageclasstopic.asp`

IMessageSink Interface

As mentioned previously, when a message is sent from an object to a remote object, it first flows through a set of sinks in the local application domain before it is sent to the remote object via the transport channel. On the remote object's side, a receiving channel receives the object and passes the message through its own chain of message sinks again before the last sink decodes the message and transforms the message into a local method call to the actual remote object's method or property.

The IMessageSink interface defines the basic functionality for every message sink of the framework.

Interface definition:

```
public interface IMessageSink
{
 IMessageCtrl AsyncProcessMessage(IMessage msg, IMessageSink replySink);
 IMessage SyncProcessMessage(IMessage msg);
 IMessageSink NextSink { get; }
}
```

As you can see, IMessageSink defines one property and two methods. The property, `NextSink`, returns a reference to the next message sink in the chain of sinks, whereas the two methods are used for processing the messages (one for synchronous and the other one for asynchronous processing).

Chapter references:

- Chapter 4: Configuration and Deployment
- Chapter 11: Inside the Framework
- Chapter 12: Creation of Sinks
- Chapter 13: Extending .NET Remoting
- Chapter 14: Developing a Transport Channel
- Chapter 15: Context Matters

More information on MSDN:
`http://msdn.microsoft.com/library/en-us/cpref/html/frlrfsystemruntimeremotingmessagingimessagesinkclasstopic.asp`

IMethodMessage Interface

Whereas the IMessage interface just defines the basic structure of any message sent across the wire, the IMethodMessage interface is an extension of the IMessage interface that defines additional properties for messages encapsulating method-general properties. This information is sent to and from remote methods.

Interface definition:

```
public interface IMethodMessage : IMessage
{
 object GetArg(int argNum);
 string GetArgName(int index);
 int ArgCount { get; }
 object[] Args { get; }
 bool HasVarArgs { get; }
 LogicalCallContext LogicalCallContext { get; }
 MethodBase MethodBase { get; }
 string MethodName { get; }
 object MethodSignature { get; }
 string TypeName { get; }
 string Uri { get; }
}
```

The interface defines methods and properties for resolving the cracking the method affected by the call: arguments (count, name, and values), method signature information (target object type, method name, object URI), as well as the context for the message call.

Chapter reference:

- Chapter 15: Context Matters

More information on MSDN:
http://msdn.microsoft.com/library/en-us/cpref/html/frlrfsystemruntimeremotingmessagingimethodmessageclasstopic.asp

IMethodCallMessage Interface

While the IMethodMessage interface defines the basic structure for referencing methods in general and is sent to and from remote methods, IMethodCallMessage is just used for calling only remote messages.

Interface definition:

```
public interface IMethodCallMessage : IMethodMessage
{
 object GetInArg(int argNum);
 string GetInArgName(int index);
 int InArgCount { get; }
 object[] InArgs { get; }
}
```

As it is used for calling remote methods only, the interface defines additional properties for input arguments, whereas the IMethodMessage interface's properties are defined for input as well as the output argument (that's the big difference).

Chapter reference:

- Chapter 13: Extending .NET Remoting

More information on MSDN:
http://msdn.microsoft.com/library/en-us/cpref/html/frlrfsystemruntimeremotingmessagingimethodcallmessageclasstopic.asp

IMethodReturnMessage Interface

Whereas IMethodCallMessage defines the message structure sent across the wire for calling remote methods, IMethodReturnMessage defines the message structure sent back as the result of the remote message call from the server.

Interface definition:

```
public interface IMethodReturnMessage : IMethodMessage
{
 object GetOutArg(int argNum);
 string GetOutArgName(int index);
 Exception Exception { get; }
 int OutArgCount { get; }
 object[] OutArgs { get; }
 object ReturnValue { get; }
}
```

The IMethodReturnMessage interface is derived from IMethodMessage and provides additional properties for retrieving exceptions and output arguments, as well as the return value of the method call. Again, the IMethodMessage interface's properties are defined for input as well as output arguments, whereas these properties are for output arguments (and return values) only.

More information on MSDN:
`http://msdn.microsoft.com/library/en-us/cpref/html/frlrfsystemruntimeremotingmessagingimethodreturnmessageclasstopic.asp`

MethodCall Class

The MethodCall class is an implementation of the IMethodCallMessage interface and therefore provides an implementation for calling methods on a remote object. This type is not intended to be used directly in your application, therefore always use the interfaces described previously if you want to access details about method calls.

Chapter references:

- Chapter 11: Inside the Framework
- Chapter 13: Extending .NET Remoting

More information on MSDN:
`http://msdn.microsoft.com/library/en-us/cpref/html/frlrfsystemruntimeremotingmessagingmethodcallclasstopic.asp`

MethodResponse Class

The MethodResponse class implements the IMethodResponseMessage interface and therefore is the counterpart of the MethodCall class. An instance of this class is returned from the server as a result of a remote method call.

Again, this class implements the interfaces described previously in this appendix and is not intended to be used directly. Therefore, if you want to access details about the response of a remote method call, use the IMethodResponseMessage interface.

More information on MSDN:
`http://msdn.microsoft.com/library/en-us/cpref/html/frlrfsystemruntimeremotingmessagingmethodresponseclasstopic.asp`

System.Runtime.Remoting.Activation

In the System.Runtime.Remoting.Activation namespace, you can find classes and interface definitions that support activation of remote objects. Activation messages are only necessary for client-activated objects, as server-activated objects are created by the server itself.

Basically, the creation of remote objects is based on so-called construction call messages. These messages are special implementations of IMethodCallMessage as well as IMethodResponseMessage.

IConstructionCallMessage Interface

This interface defines the structure of the message sent for activating a client-activated object of the server. Therefore, when the client calls `Activator.CreateInstance()` or uses the `new` operator for creating a configured client-activated object, the .NET Remoting infrastructure creates a construction call message and sends this message to the server for retrieving, telling the server to create a new instance as well as retrieving the reference to the newly created instance.

Interface definition:

```
public interface IConstructionCallMessage : IMethodCallMessage
{
 Type ActivationType { get; }
 string ActivationTypeName { get; }
 IActivator Activator { get; set; }
 object[] CallSiteActivationAttributes { get; }
 IList ContextProperties { get; }
}
```

The construction message contains the actual type of the remote object to be created and the name of the type, as well as context information and activation attributes specified in the `Activator.CreateInstance()` method call on the client. The `Activator` property specifies an instance of IActivator, which is responsible for the creation process.

Chapter reference:

- Chapter 15: Context Matters

More information on MSDN:
`http://msdn.microsoft.com/library/en-us/cpref/html/frlrfsystemruntimeremotingactivationiconstructioncallmessageclasstopic.asp`

IConstructionReturnMessage Interface

The IConstructionReturnMessage interface is an implementation of IMethodReturnMessage and provides the client with results of the activation of a client-activated object. Therefore, it is used by the infrastructure for checking the success of the creation process as well as retrieving context information created by the activator during the creation process.

IConstructionReturnMessage defines no additional properties or methods in the current version of the .NET Framework.

More information on MSDN:
`http://msdn.microsoft.com/library/en-us/cpref/html/frlrfsystemruntimeremotingactivationiconstructionreturnmessageclasstopic.asp`

System.Runtime.Remoting.Proxies

The System.Runtime.Remoting.Proxies namespace contains the classes for controlling and providing proxy functionality to the .NET Remoting framework. The most important class within this namespace is the RealProxy class, which is the base class for all client-side proxies. You can use this class for creating custom proxies, too.

RealProxy Class

RealProxy is an abstract base class for all client-side proxies. Actually, the client never uses RealProxy directly; it always has access to a TransparentProxy. TransparentProxy gives the client the illusion of talking to the remote object directly. Actually, TransparentProxy forwards any method calls to a RealProxy instance.

RealProxy has the task of taking the method calls from TransparentProxy and forwards these method calls through the .NET Remoting infrastructure to the remote server object.

Chapter references:

- Chapter 11: Inside the Framework
- Chapter 12: Creation of Sinks
- Chapter 13: Extending .NET Remoting

More information on MSDN:
`http://msdn.microsoft.com/library/en-us/cpref/html/frlrfsystemruntimeremotingproxiesrealproxyclasstopic.asp`

ProxyAttribute Class

This attribute indicates that an object type requires a custom proxy object inherited from the RealProxy class. Any object that requires a custom proxy needs to have this class-level attribute applied.

Usually, you would have to provide your own implementation of this attribute. The attribute employs a `CreateProxy()` method in which you can create, initialize, and return an instance of your custom proxy object.

More information on MSDN:
`http://msdn.microsoft.com/library/en-us/cpref/html/frlrfsystemruntimeremotingproxiesproxyattributeclasstopic.asp`

System.Runtime.Remoting.Channels

Channels and channel sinks are used as transport vehicles when a client calls methods on a remote object. The System.Runtime.Remoting.Channels namespace contains classes as well as base interfaces for those classes as well as other classes in subnamespaces.

The most important classes are described in Appendix A. Here, I will focus on the interfaces that are used for extending the .NET Remoting infrastructure. The interfaces in this namespace are used for both creation of custom transport channels and extension of the framework by creating custom sinks.

When a message is sent to a remote object, the message first flows through a chain of so-called message sinks (remember the IMessageSink interface introduced in the previous section of this chapter). The sink chain must consist of one formatter sink that is a special sink that creates the wire format of the message. Afterwards, some preprocessing sinks (like encryption or digital signatures) process the message before passing it on to the last sink in the chain, which must be the transport channel sink itself.

On the server, the processing occurs the other way around. The transport channel is the first sink that receives the message from the client. It then forwards the message to some preprocessing sinks (decryption, digital signature verification) as well as formatter sinks and other (custom) sinks before cracking the message into its parts and converting it to a method call on the server's object.

For both client- and server-side processing, as well as sending and receiving parts, you will find the necessary interfaces for customizing this sink chain in the namespace.

IChannelSinkBase Interface

IChannelSinkBase is the base interface for all types of sinks in the sink chain. It defines just a property bag for a channel sink. All the implementations extend the channel sink base structure with their own properties (for indirectly accessing this property bag).

When extending the .NET Remoting infrastructure, you don't use this interface, but the interfaces described in the following parts of this section.

Chapter references:

- Chapter 13: Extending .NET Remoting
- Chapter 14: Developing a Transport Channel

More information on MSDN:

`http://msdn.microsoft.com/library/en-us/cpref/html/frlrfsystemruntimeremotingchannelsichannelsinkbaseclasstopic.asp`

IClientChannelSink Interface

IClientChannelSink defines the basic functionality for client-side channel sinks and therefore defines which functionality must be provided by a custom plug-in point within the client-side message processing.

Interface definition:

```
public interface IClientChannelSink : IChannelSinkBase
{
 void AsyncProcessRequest(IClientChannelSinkStack sinkStack,
                          IMessage msg, ITransportHeaders headers,
                          Stream stream);
 void AsyncProcessResponse(IClientResponseChannelSinkStack sinkStack,
                           object state, ITransportHeaders headers,
                           Stream stream);
 Stream GetRequestStream(IMessage msg, ITransportHeaders headers);
 void ProcessMessage(IMessage msg, ITransportHeaders requestHeaders,
                     Stream requestStream,
                     [out] ref ITransportHeaders responseHeaders,
                     [out] ref Stream responseStream);
 IClientChannelSink NextChannelSink { get; }
}
```

The interface defines a property for linking the current sink to the next sink in the sink chain. The `ProcessMessage()` method is used for processing a message request and its response synchronously. This means that `ProcessMessage()` processes the request, calls the next sink in the chain (its `ProcessMessage()` method), and waits till the sink (or the remote object) has finished processing.

For asynchronous calls, the interface defines a method for asynchronously processing the requests without waiting for the response and for asynchronously processing the response as soon as it is available. Through the `GetRequestStream()` method, the sink has direct access to the stream onto which the provided message will be serialized.

Chapter references:

- Chapter 11: Inside the Framework
- Chapter 12: Creation of Sinks
- Chapter 13: Extending .NET Remoting
- Chapter 14: Developing a Transport Channel

More information on MSDN:
`http://msdn.microsoft.com/library/en-us/cpref/html/frlrfsystemruntimeremotingchannelsiclientchannelsinkclasstopic.asp`

IClientChannelSinkProvider Interface

Sink implementations are always connected to channels through sink providers. This can also be seen in configuration files where you always configure new sinks through the <provider> tag and specify corresponding sink provider classes.

A sink provider can be seen as the factory for a sink implementation itself. This means the sink provider is responsible for creating, initializing, and returning the actual channel sink instance.

Interface definition:

```
public interface IClientChannelSinkProvider
{
 IClientChannelSink CreateSink(IChannelSender channel,
                               string url, object remoteChannelData);
 IClientChannelSinkProvider Next { get; set; }
}
```

The interface defines one method for creating the actual sink as well as a property for setting and retrieving the next sink provider in the chain. This means the developer of a sink is responsible for calling the `CreateSink()` method of the next provider (if available) before returning its own sink instance within the `CreateSink()` method implementation.

Chapter references:

- Chapter 12: Creation of Sinks
- Chapter 13: Extending .NET Remoting
- Chapter 14: Developing a Transport Channel

More information on MSDN:
http://msdn.microsoft.com/library/en-us/cpref/html/frlrfsystemruntimeremotingchannelsiclientchannelsinkproviderclasstopic.asp

IClientFormatterSink Interface

IClientFormatterSink is a special interface implementing the IMessageSink interface as well as the IClientChannelSink interface. Formatters are special sinks in the sink chain responsible for formatting the message into its wire format before sending it across the wire.

The interface doesn't define any additional methods. It just combines the interfaces IMessageSink and IClientChannelSink within one interface. The first sink on the client side must be an IClientFormatterSink or implement both the IMessageSink and the IClientChannelSink interfaces.

In configuration files, you usually use the <formatter> tag instead of the <provider> tag for specifying a formatter in the sink chain.

Chapter reference:
- Chapter 11: Inside the Framework

More information on MSDN:
http://msdn.microsoft.com/library/en-us/cpref/html/frlrfsystemruntimeremotingchannelsiclientformattersinkclasstopic.asp

IClientFormatterSinkProvider Interface

This interface marks a sink provider as a provider used for creating formatter sink objects. The interface doesn't add any methods to its base interface IClientChannelSinkProvider, it is just used as a marker for the .NET Remoting runtime.

The first sink provider in the chain must be a formatter sink provider. Use the <formatter> tag instead of the <provider> tag in the configuration file for specifying the client-side formatter.

Sink formatter implementations usually use the runtime serialization formatters (BinaryFormatter or SoapFormatter) specified in the System.Runtime.Serialization namespace as well as in the Formatters subnamespace.

More information on MSDN:
http://msdn.microsoft.com/library/en-us/cpref/html/frlrfsystemruntimeremotingchannelsiclientformattersinkproviderclasstopic.asp

IServerChannelSink Interface

Whereas the IClientChannelSink interface is used for creating sinks employed before a message is sent across the wire to a remote object, you can use the IServerChannelSink interface for creating sinks used when a remote object receives a message from a client. That said, this interface defines the functionality that has to be supported by server-side sink objects.

Interface definition:

```
public interface IServerChannelSink : IChannelSinkBase
{
 void AsyncProcessResponse(IServerResponseChannelSinkStack sinkStack,
                           object state, IMessage msg,
                           ITransportHeaders headers, Stream stream);
 Stream GetResponseStream(IServerResponseChannelSinkStack sinkStack,
                          object state, IMessage msg, ITransportHeaders headers);
 ServerProcessing ProcessMessage(IServerChannelSinkStack sinkStack,
                                 IMessage requestMsg,
                                 ITransportHeaders requestHeaders,
                                 Stream requestStream,
                                 [out] ref IMessage responseMsg,
                                 [out] ref ITransportHeaders responseHeaders,
                                 [out] ref Stream responseStream);
 IServerChannelSink NextChannelSink { get; }
}
```

Although the interface is an extension of the IChannelSinkBase interface like its client-side counterpart, its structure is a little bit more complicated. Most importantly, it defines a method for synchronously processing incoming messages—the `ProcessMessage()` method.

For asynchronous messaging, it specifies the `AsyncProcessResponse()` method only as the logic for message processing, and not waiting for any response is encapsulated within the `ProcessMessage()` method itself.

Whereas the IClientChannelSink interface allows you to access the request stream, the server channel sink requires a method for retrieving the response stream into which the returned message is going to be serialized. And last but not least, the concept for accessing the next sink in the chain through the `NextChannelSink` property is still the same as with client-side channel sinks.

Chapter references:

- Chapter 11: Inside the Framework
- Chapter 12: Creation of Sinks
- Chapter 13: Extending .NET Remoting
- Chapter 14: Developing a Transport Channel

More information on MSDN:

`http://msdn.microsoft.com/library/en-us/cpref/html/frlrfsystemruntimeremotingchannelsiserverchannelsinkclasstopic.asp`

IServerChannelSinkProvider Interface

The IServerChannelSinkProvider interface acts as a factory for creating server-side channel sinks. This concept is basically the same as with the client-side sink provider classes. In configuration files, you don't specify any server channel sink directly but rather its sink providers.

Interface definition:

```
public interface IServerChannelSinkProvider
{
 IServerChannelSink CreateSink(IChannelReceiver channel);
 void GetChannelData(IChannelDataStore channelData);
 IServerChannelSinkProvider Next { get; set; }
}
```

Although fulfilling the same purpose for server sinks as IClientChannelSinkProvider for client-side sinks, the interface is differently structured. Remember that the server side doesn't need the object URI because it is receiving messages only (sending processes will be done through the client sink infrastructure because that means the server plays the role of a client when communicating with another server object). Also, it doesn't need the possibility for specifying additional data that will be sent to the remote channel (that is, the remoteChannelData parameter in IClientChannelSinkProvider's `CreateSink()` method).

Through the `GetChannelData()` method, the sink provider is able to access channel specific properties of the receiving channel. The `Next` property is used by the infrastructure for setting the next server sink provider in the chain and enables you to retrieve the next provider in your code.

Again, you are responsible for calling the `CreateSink()` method of the next channel sink provider before returning your own sink provider instance.

Chapter references:

- Chapter 12: Creation of Sinks
- Chapter 13: Extending .NET Remoting
- Chapter 14: Developing a Transport Channel

More information on MSDN:

`http://msdn.microsoft.com/library/en-us/cpref/html/frlrfsystemruntimeremotingchannelsiserverchannelsinkproviderclasstopic.asp`

ITransportHeaders Interface

When discussing the IMessage interface, I told you that a message contains both metadata and the actual user data. Whereas the user data is the part necessary for implementing the business logic, metadata can be used for some infrastructural additional information.

For example, if you are implementing an asynchronous encryption channel, information about the public key can be included in the transport headers collection. This information doesn't really belong to the user data (e.g., invoice, order, or similar business messages), but it is used by (your own) infrastructure for providing some basic services. Such information—called metadata—can be stored in the transport headers of a message.

The ITransportHeaders interface defines the basic functionality for transport header objects. It is just a collection of key-value pairs, as you can see in its interface definition.

Interface definition:

```
public interface ITransportHeaders
{
 IEnumerator GetEnumerator();
object this[object key] { get; set; }
}
```

The interface defines a `GetEnumerator()` method, which returns an enumerator for iterating the header properties as well as an indexer property for accessing the header properties directly by a key.

Chapter references:

- Chapter 11: Inside the Framework
- Chapter 13: Extending .NET Remoting
- Chapter 14: Developing a Transport Channel

More information on MSDN:
`http://msdn.microsoft.com/library/en-us/cpref/html/frlrfsystemruntimeremotingchannelsitransportheadersclasstopic.asp`

IChannel Interface

The IChannel interface defines the basic functionality of a transport channel. Transport channels are used for crossing remoting boundaries, which can be contexts, AppDomains, and processes, as well as machines.

Channels implement the specifics of the transport protocol (e.g., TCP or HTTP) and can listen on transport protocol ports as well as send messages through transport protocol streams.

That said, channels have to cover inbound as well as outbound communication with remote objects. They provide an extensibility point in the runtime for adding custom transport protocols to the .NET Remoting infrastructure.

Interface definition:

```
public interface IChannel
{
 string Parse(string url, [out] ref string objectURI);
 string ChannelName { get; }
 int ChannelPriority { get; }
}
```

As you can see, each channel has to implement a method for parsing the remote object's URL as well as provide properties for the channel's name and its priority in the list of channels. Both roles, the listening portion as well as the sending portion, are expressed through the subinterfaces IChannelReceiver and IChannelSender described in the next two sections of this appendix.

Chapter references:

- Chapter 4: Configuration and Deployment
- Chapter 12: Creation of Sinks
- Chapter 13: Extending .NET Remoting
- Chapter 14: Developing a Transport Channel

More information on MSDN:
`http://msdn.microsoft.com/library/en-us/cpref/html/frlrfsystemruntimeremotingchannelsichannelclasstopic.asp`

IChannelReceiver Interface

The IChannelReceiver interface, an extension of the IChannel interface, defines the functionality that has to be provided for receiving channels. This means the receiver listens on a specific transport protocol port and waits for incoming messages. Every time a message is received, it takes the message and passes it on to the formatter, which deserializes the message and forwards it to the next sink in the sink chain.

That said, IChannelReceiver always has to be the first part in the sink chain of a remoting application that is able to receive messages from remote objects.

Interface definition:

```
public interface IChannelReceiver : IChannel
{
 string[] GetUrlsForUri(string objectURI);
 void StartListening(object data);
 void StopListening(object data);
 object ChannelData { get; }
}
```

Usually, a receiving channel is used on the server side, waiting for incoming requests of clients. If you keep configuration in mind, you are specifying an object URI only and not the whole URL. Therefore, the receiving channel needs to provide functionality for creating the real URL out of the specified object URI, which is encapsulated in the `GetUrlsForUri()` method of the interface.

Furthermore, the channel knows how to start listening as well as stop listening on the transport protocols port, and therefore needs to provide functionality for doing so. The `ChannelData` property provides access to additional channel properties.

Chapter references:

- Chapter 4: Configuration and Deployment
- Chapter 12: Creation of Sinks
- Chapter 13: Extending .NET Remoting
- Chapter 14: Developing a Transport Channel

More information on MSDN:
http://msdn.microsoft.com/library/en-us/cpref/html/frlrfsystemruntimeremotingchannelsichannelreceiverclasstopic.asp

IChannelSender Interface

The IChannelSender interface is the counterpart of the IChannelReceiver interface and specifies functionality a channel has to support when sending messages to a remote remoting channel.

Interface definition:

```
public interface IChannelSender : IChannel
{
 IMessageSink CreateMessageSink(string url,
                                object remoteChannelData,
                                [out] ref string objectURI);
}
```

Whereas IChannelReceiver actually is responsible for listening on the transport protocols port and receiving the messages through this port, the channel sender is just responsible for creating a message sink that does the actual protocol handling.

This sink will be added to the sink chain of the infrastructure on the last position. This means, as the last sink, it is responsible for sending the message via the corresponding transport protocol across the wire.

For example, both HttpClientChannel and TcpClientChannel are implementations of this interface but do not send the message across the wire themselves. They act as a factory for an HttpClientTransport sink and TcpClientTransportSink, respectively. Those sinks are actually sending the message across the wire to the remote endpoint.

Chapter references:

- Chapter 4: Configuration and Deployment
- Chapter 12: Creation of Sinks
- Chapter 13: Extending .NET Remoting
- Chapter 14: Developing a Transport Channel

More information on MSDN:
http://msdn.microsoft.com/library/en-us/cpref/html/frlrfsystemruntimeremotingchannelsichannelsenderclasstopic.asp

BaseChannelObjectWithProperties Class

This class provides a base implementation for channels and channel sinks that want to provide properties. It implements all necessary interfaces for providing those properties in the form of key-value pairs and can be used as a base class for channels or channel sink objects.

The class does not implement any of the interfaces described previously. It only implements IDictionary, IEnumerable, and ICollection. Therefore, the sink interfaces or channel interfaces must be implemented manually. The class primarily handles the task of asking a channel for its properties.

More information on MSDN:
http://msdn.microsoft.com/library/en-us/cpref/html/frlrfsystemruntimeremotingchannelsbasechannelobjectwithpropertiesclasstopic.asp

BaseChannelWithProperties Class

The BaseChannelWithProperties class can be used as a base class for implementing your own channels. It extends BaseChannelObjectWithProperties with the complex task of asking its channel sinks for their properties.

Still, this interface does not implement any of the channel-specific interfaces introduced in the previous parts of this appendix (e.g., IChannel, IChannelSender, or IChannelReceiver); therefore, you have to provide your own implementation of these interfaces.

Chapter references:

- Chapter 13: Extending .NET Remoting
- Chapter 14: Developing a Transport Channel

More information on MSDN:
http://msdn.microsoft.com/library/en-us/cpref/html/frlrfsystemruntimeremotingchannelsbasechannelwithpropertiesclasstopic.asp

BaseChannelSinkWithProperties Class

This class is an extension of BaseChannelObjectWithProperties that can be used as a base class for implementing your own channel sink provider. As it doesn't implement any of the interfaces described in the previous sections (e.g., IClientChannelSinkProvider) you have to implement these interfaces on your own.

Chapter references:

- Chapter 13: Extending .NET Remoting
- Chapter 14: Developing a Transport Channel

More information on MSDN:
http://msdn.microsoft.com/library/en-us/cpref/html/frlrfsystemruntimeremotingchannelsbasechannelsinkwithpropertiesclasstopic.asp

Summary

With this appendix, you have received a collection of reference information that you will need when extending the .NET Remoting framework. I've included most interfaces and base classes you'll need to create your own message sinks, channel sinks, and transport channels.

In the next appendix, I've collected a series of links for .NET Remoting. You'll find implementations of all the interfaces described in this chapter.

APPENDIX C

.NET Remoting Links

This last appendix provides a list of useful links as well as short descriptions of the content that can be found on the target locations of those links. Most of the URLs listed here are links to some technical articles and how-to articles as well as useful and interesting samples.

Ingo's .NET Remoting FAQ Corner

On the homepage of thinktecture, you will find a separate .NET Remoting corner written by one of the authors of this book. Here he publishes technical articles as well as important links to information, knowledge base articles, and other technical resources for .NET Remoting. A must-know for anyone who programs solutions based on .NET Remoting.

`http://www.thinktecture.com/Resources/RemotingFAQ/default.html`

MSDN and *MSDN Magazine* Articles

In the past three years, Microsoft has published a lot of in-depth information about .NET Remoting. Following is a rundown of some of the more informative ones.

"Improving Remoting Performance"

This article is part of the Patterns & Practices book *Improving .NET Application Performance and Scalability* (Microsoft Press, 2004). It presents concrete recommendations for when to use remoting and when not to use it, together with appropriate alternatives. The design guidelines and coding techniques in this chapter provide performance solutions for activation, channels, formatters, and serialization.

`http://msdn.microsoft.com/library/en-us/dnpag/html/scalenetchapt11.asp`

".NET Remoting Security"

This article is part of the Patterns & Practices book *Building Secure ASP.NET Applications* (Microsoft Press, 2004). It describes how to implement authentication, authorization, and secure communication in distributed Web applications that use .NET Remoting.

`http://msdn.microsoft.com/library/en-us/secmod/html/secmod11.asp`

"Boundaries: Processes and Application Domains"

As soon as you have to communicate between two assemblies loaded into different application domains, you must use .NET Remoting. Application domains are a new concept introduced with the .NET Framework for isolating .NET assemblies loaded into the same operating system process.

ASP.NET, for example, uses this mechanism on IIS 5.x-based machines for isolating ASP.NET Web applications, and SQL Server 2005 uses this mechanism for isolating assemblies installed in different databases. More information about application domains can be found here:

```
http://msdn.microsoft.com/library/en-us/cpguide/html/
cpconboundariesprocessesapplicationdomainscontexts.asp
```

".NET Remoting Architectural Assessment"

Even if you know the technical details about .NET Remoting, finding the right architecture is not very easy. This article is intended for anyone who wants to use .NET Remoting for building distributed applications. It describes the capabilities of the technology on an architectural level, as well as some interesting implications when it comes to using the different features of the infrastructure.

```
http://msdn.microsoft.com/library/en-us/dndotnet/html/dotnetremotearch.asp
```

".NET Remoting Overview"

The official tutorial for .NET Remoting consists of lots of technical articles from the very basics to some advanced topics as well as a complete reference of the .NET Remoting infrastructure.

```
http://msdn.microsoft.com/library/en-us/cpguide/html/
cpconnetremotingoverview.asp
```

"Building Secure ASP.NET Applications: Authentication, Authorization, and Secure Communication"

The Microsoft Patterns & Practices team released a comprehensive guide for building secure ASP.NET applications. It contains architectural as well as in-depth technical information for designing and writing secure ASP.NET Web applications. Although the guide focuses on Web applications, it contains a separate chapter about .NET Remoting security, because .NET Remoting becomes important when the Web front end starts talking to remote back-end components. This chapter can be found here:

```
http://msdn.microsoft.com/library/en-us/dnnetsec/html/SecNetch11.asp
```

"NET Remoting Authentication and Authorization Sample"

Chapter 5 introduces a solution for implementing authentication and secure communication using .NET Remoting. In .NET 2.0, the infrastructure is included in the .NET Framework itself, but in .NET 1.x you have to use the security solution sample assemblies introduced in the following two MSDN articles. More information about using the solution can be found in Chapter 5.

```
http://msdn.microsoft.com/library/en-us/dndotnet/html/remsspi.asp
http://msdn.microsoft.com/library/en-us/dndotnet/html/remsec.asp
```

"Managed Extensions for C++ and .NET Remoting Tutorial"

If you are using Visual C++ and managed extensions for C++, this tutorial might be interesting for you. It focuses on .NET Remoting when writing applications with managed C++.

```
http://msdn.microsoft.com/library/en-us/vcmex/html/
vcgrfmanagedextensionsforcnetremotingtutorial.asp
```

".NET Remoting Use-Cases and Best Practices" and "ASP.NET Web Services or .NET Remoting: How to Choose"

One of the most frequently asked questions is when to use .NET Remoting, especially when it comes to the decision of whether to use Web Services or .NET Remoting, which proves hard for most people. Well, each of the technologies mentioned previously has its advantages and targets specific use cases. The following links provide you with information that helps you in your decision process:

```
http://www.thinktecture.com/Resources/RemotingFAQ/RemotingUseCases.html
http://msdn.microsoft.com/library/en-us/dnbda/html/bdadotnetarch16.asp
```

"Remoting Examples"

In the MSDN Resource Center, you will find lots of .NET Remoting samples. The following link brings up a page with some advanced .NET Remoting samples showing the internals of the infrastructure.

```
http://msdn.microsoft.com/library/en-us/cpguide/html/
cpconRemotingExamples.asp
```

"Secure Your .NET Remoting Traffic by Writing an Asymmetric Encryption Channel"

Although the security solution mentioned previously provides you with a complete solution covering authentication and secure communication, this article of *MSDN Magazine* is interesting if you want to learn more about writing custom channel sinks as well as the classes of the System.Security.Cryptography namespace.

```
http://msdn.microsoft.com/msdnmag/issues/03/06/NETRemoting/
```

"Create a Custom Marshaling Implementation Using .NET Remoting and COM Interop"

Several methods for customizing the presentation of native .NET and COM object types are available with the .NET Framework. Custom marshaling, which is one such technique, refers to the notion of specializing object type presentations. Elements of COM Interop permit the customizing of COM types, whereas .NET Remoting offers the developer the ability to tailor native .NET types. This article examines these techniques.

```
http://msdn.microsoft.com/msdnmag/issues/03/09/custommarshaling/
```

.NET Remoting Interoperability

In my opinion, the primary technology for interoperability should be Web Services. But sometimes Web Services might not be an option, especially if you need tight coupling of applications based on different platforms (although I would avoid tight coupling as it leads to increased effort when one of the applications changes). In this case, you might be better off using a different technology that provides you with the performance and possibilities needed in your special case. Some of the following solutions provide you with interoperability between .NET and other platforms based on .NET Remoting.

.NET Remoting: CORBA Interoperability

Remoting.Corba (pronounced remoting dot corba) is a project that aims to integrate CORBA/IIOP support into the .NET Remoting architecture. The goal is to allow .NET programmers to use C# and Visual Basic .NET to develop systems that interoperate with systems that support the Internet Inter-ORB Protocol (IIOP), including CORBA systems and various application servers and middleware technologies.

```
http://remoting-corba.sourceforge.net/
```

.NET Remoting: Java RMI Bridges

Interoperability between Java and .NET becomes more and more important by now. Many large enterprises using applications based on Java as well as .NET need to integrate those applications. Although Web Services should be the primary technology because it provides the foundation for loose coupling (which makes the applications more independent of each other), in some cases you might need tight coupling (for performance reasons, stateful work, or similar things).

When it comes to interoperability with Java-based applications, you need to work with a so-called Java RMI to .NET Remoting bridge. Such bridges enable .NET-based applications using Java applications published via Java RMI and vice versa. The most common bridges, J-Integra for .NET and JNBridge Pro, are available at the following URLs:

```
http://j-integra.intrinsyc.com/net/info/
http://www.jnbridge.com/jnbpropg.htm
```

XML-RPC with .NET Remoting

XML-RPC.NET is a library for implementing XML-RPC services and clients in the .NET environment. The library has been in development since March 2001 and is used in many open source and business applications.

```
http://www.xml-rpc.net/
```

Custom .NET Remoting Channels

Out-of-the-box, up to version 1.1 of the .NET Framework, .NET Remoting comes with two channels: TcpChannel and HttpChannel. In version 2.0 of the .NET Framework, the IpcChannel will be added. In many cases, other channels might be more appropriate than these two. Here you can find a list of interesting channels as well as some hints for when it is useful to use them.

Named Pipes Channel for .NET Remoting

Both TcpChannel and HttpChannel are optimized for communication between two machines across the network. If you just want to implement interprocess communication, they include unnecessary overhead. For this purpose, named pipes are more appropriate. With .NET Framework 2.0, a new channel, IpcChannel, will be included in the .NET Remoting runtime (for more information, see Chapter 4).

With .NET Framework 1.x, you have to use a custom implementation. Such an implementation can be found at GotDotNet:

```
http://www.gotdotnet.com/Community/UserSamples/Details.aspx?SampleGuid=
43a1ef11-c57c-45c7-a67f-ed68978f3d6d
```

TcpEx Channel for .NET Remoting

The TcpEx channel is a replacement for the built-in TCP remoting channel. It improves on the standard TCP channel by allowing communication in both directions on a single TCP connection, instead of opening a second connection for events and callbacks.

```
http://www.gotdotnet.com/Community/UserSamples/Details.aspx?SampleGuid=
3F46C102-9970-48B1-9225-8758C38905B1
```

Jabber Channel

This channel allows you to transfer .NET Remoting messages via the Jabber instant messaging protocol.

```
http://www.thinktecture.com/Resources/Software/opensourceremoting/
JabberChannel.html
```

Remoting Channel Framework Extension

Implementing new transport channels can be a heavy challenge. The Remoting Channel Framework Extension introduced on GotDotNet provides you with some additional classes based on the existing .NET Remoting infrastructure that makes development of transport channels easier.

```
http://www.gotdotnet.com/Community/UserSamples/Details.aspx?SampleGuid=
3c35d911-e138-4ce9-87bc-47f0525ca203
```

"Using MSMQ for Custom Remoting Channel"

This article, found at the following Code Project page, describes the design and development of a custom channel for using MSMQ with .NET Remoting. The implementation provides you with an MSMQ sender as well as receiver channel.

```
http://www.codeproject.com/csharp/msmqchannel.asp
```

"Using WSE-DIME for Remoting over Internet"

DIME is a standard for transferring binary data together with a SOAP message between a Web Service client and the Web Service itself. This article describes a design and implementation of remoting over the Internet using the Advanced Web Services Enhancements—DIME technology. This solution allows the binary formatted Remoting messages that flow through the Web Server to include uploading and downloading any types of the attachments.

`http://www.codeproject.com/cs/webservices/remotingdime.asp`

Interesting Technical Articles

Some additional interesting articles can be found on the Web. The following text describes a few of them that you may find especially interesting.

C# Corner: Remoting Section

The C# Corner community is a member of the CodeWise community and offers lots of technical articles and samples. They also have a separate corner for .NET Remoting that contains articles and samples specific to this topic.

`http://www.c-sharpcorner.com/Remoting.asp`

"Share the ClipBoard Using .NET Remoting"

This article describes how to use .NET Remoting for sharing the clipboard of one computer with other computers.

`http://www.codeproject.com/dotnet/clipsend.asp`

"Chaining Channels in .NET Remoting"

This article describes how to design and implement remoting over chained channels (standard and custom) using the logical URL address connectivity.

`http://www.codeproject.com/csharp/chainingchannels.asp`

"Applying Observer Pattern in .NET Remoting"

Implementing an observer pattern in distributed applications can be quite challenging. But this article gives you a very good overview of how to design and implement the observer pattern with .NET Remoting–based applications.

`http://www.codeproject.com/csharp/c_sharp_remoting.asp`

"Aspect-Oriented Programming Enables Better Code Encapsulation and Reuse" and ".NET Remoting Spied On"

Method call interception is interesting when it comes to aspect-oriented programming where you add aspects to objects and methods by just using attributes. The attributes are used by an infrastructure (framework, base components, for example, COM+) and lead to some infrastructural tasks. The following two articles describe the possibilities offered by the .NET Remoting infrastructure for method call interception, which is necessary for implementing this approach.

```
http://msdn.microsoft.com/msdnmag/issues/02/03/aop/
http://www.codeproject.com/dotnet/remotespy.asp
```

"Persistent Events in Stateless Remoting Server"

High availability in distributed applications usually requires you to implement stateless components on the server. This article shows the usage of .NET events with stateless server components:

```
http://www.codeproject.com/csharp/persistentevents.asp
```

"Intrinsyc's Ja.NET—Extending the Reach of .NET Remoting"

Java RMI to .NET Remoting bridges enable interoperability between Java-based and .NET-based applications. This article describes how to use Ja.NET, a Java RMI to .NET Remoting bridge. Find the link to the Ja.NET bridge earlier in this chapter in the section ".NET Remoting: Java RMI Bridges."

```
http://www.devx.com/dotnet/Article/6973
```

"Implementing Object Pooling with .NET Remoting—Part I"

This article shows an interesting way for implementing object pooling with custom resources and objects when using the .NET Remoting infrastructure together with the ITrackingHandler interface and tracking services.

```
http://www.devx.com/vb2themax/Article/19895/0/page/1
```

".NET Remoting Versus Web Services"

The decision for the technology in distributed application scenarios is not easy and gets even harder with the number of possibilities for implementation. The most common frequently asked question, of course, is whether to use Web Services or .NET Remoting in distributed applications. This article might help you with your decision, but don't forget about the articles mentioned earlier in this appendix.

```
http://www.developer.com/net/net/article.php/11087_2201701_1
```

".NET Remoting Central"

This provides a collection of links for samples and information about .NET Remoting.

```
http://www.dotnetpowered.com/remoting.aspx
```

"Output Caching for .NET Remoting"

ASP.NET output caching can be used with not only ASP.NET pages, but also any other type of application hosted in IIS, too. In this workspace, you can find a server-side channel sink that provides output caching of methods on remoted objects.

http://www.gotdotnet.com/Community/UserSamples/Details.aspx?SampleGuid=6a228851-5479-46bb-a972-6ad4b50d870e

"Abstract Client Formatter Sink"

You may want to connect to more than one remote server over a TCP channel using a binary formatter in one case and a SOAP formatter in the other case. This would not work with the normal remoting system, but this project demonstrates a workaround. A document is included with the project explaining all of it better.

http://www.gotdotnet.com/Community/UserSamples/Details.aspx?SampleGuid=0c348249-7823-406b-9fa5-b633b8d1c579

Remoting Tools

Apart from the articles and samples just discussed, a number of tools have been developed that may help you during debugging and troubleshooting.

Remoting Management Console

The Remoting Management Console is a MMC snap-in that allows you to configure host processes for publishing .NET Remoting components. Its user interface is similar to the configuration MMC for the COM+ catalog.

http://www.codeproject.com/csharp/remotingmanagementconsole.asp

Remoting Probe

The Remoting Probe tool consists of a custom channel sink as well as an analyzer tool for analyzing the communication details between remoting objects. The channel sink is responsible for publishing communication steps to the analyzer tool, and the tool can be used for creating reports about communication.

http://www.codeproject.com/csharp/remotingprobe.asp

Index

A

ACL (Access Control Lists), 75
activated tag, client tag, 99
activated tag, service tag, 97
ActivatedClientTypeEntry class, 494–495
ActivatedServiceTypeEntry class, 494
Activation namespace, 529–530
Activator class, 19, 488
AddAuthenticationEntry method, 402, 407, 409
AddRef method, 185
AddValue method, 243, 261
algorithms
  symmetric encryption, 376
anonymous access
  ASP.NET application impersonating client, 183
  IIS authentication modes, 133–134
AOP (aspect-oriented programming)
  links to web sites, 547
application configuration files
  creating console clients, 163
application design/development
  designing applications for static scalability, 299
  versioning, 256, 273
application domains
  cross-AppDomain remoting, 276
  links to web sites, 542
  serializable classes passed between, 492
Application General code
  creating Windows Forms client, 167
application tag, 85, 165
Application_StartUp event, 175
appSettings tag, configuration tag, 165
architecture
  .NET Remoting, 10, 322
  links to web sites, 542
  sample remoting application, 14
  Service-Oriented Architecture, 279
Args property, messages, 327
ASMX Web Services, 279
ASP.NET based clients, 169, 170
aspect-oriented programming (AOP)
  links to web sites, 547
assemblies
  .NET Framework versioning, 228, 232
  client assembly, 14
  CLR locating, 227
  creating strongly named assembly, 226
  full assembly name, 225
  GAC, 227, 228
  general assembly, 14
  generated metadata assembly, 12
  private assemblies, 227
  server assembly, 14
  shared assemblies, 11, 67, 227
assemblyBinding tag, 241
AssemblyKeyFile attribute, 227, 228
AssemblyVersion attribute, 227, 237
asymmetric encryption channel
  links to web sites, 543
asynchronous calls
  client assembly, 53
  client-side sinks, 364, 459
  delegates, 51, 52
  mapping, 421
  server-side sinks, 366
  ways of executing methods, 46
  ways of executing methods in .NET, 51
asynchronous delegates, 434, 441
asynchronous messaging, 338–348
  generating requests, 342–344
  handling responses, 345–346
  IClientChannelSink processing, 340–342
  IMessageSink processing, 338–339
  mapping protocols to .NET Remoting, 440, 441
  server-side asynchronous processing, 347–348
AsyncProcessMessage method
  asynchronous IMessageSink processing, 338
  checking parameters in IMessageSink, 480

creating client-side sink and provider, 451
IMessageSink, 328
passing runtime information, 391, 393
server-side asynchronous processing, 347
AsyncProcessRequest method
asynchronous IClientChannelSink processing, 340, 341
BinaryFormatter version mismatch, 411
changing sinks programming model, 405
client-side sinks, 364
extending compression sinks, 371, 372
generating asynchronous request, 342
AsyncProcessResponse method
asynchronous IClientChannelSink processing, 340, 341
client-side sinks, 364
creating, 453
extending compression sinks, 372, 373
generating asynchronous request, 343
handling asynchronous response, 345, 346
IServerChannelSink, 334, 535
passing runtime information, 396
server-side asynchronous processing, 347
server-side sinks, 365, 367
creating, 458, 460, 461
AsyncResponseHandler class, 451, 453
AsyncResult class, 512
attributes
activated tag, 97, 99
channel tag, 89, 90
custom attributes, 92
formatter tag, 92
lifetime tag, 88
moving constraints to metadata level, 471
OneWayAttribute, 514
provider tag, 92
ProxyAttribute, 531
SoapAttribute, 515
SoapFieldAttribute, 515
SoapMethodAttribute, 516
SoapParameterAttribute, 516
SoapTypeAttribute, 515
wellknown tag, 97
authentication
see also security
anonymous access, 134
authentication level client and server, 144
authentication with IIS, 133–136
logon process, 137
basic authentication, 134
brief description, 123
changing default remoting behavior, 359
changing sinks programming model, 409
configuring authentication methods, 118
deployment using IIS, 116
deserialization of object security, 93
digest authentication, 134
encryption and IIS, 138
forms authentication, 130
identities, 130
IIS authentication modes, 133
links to web sites, 542
Passport authentication, 130
principals, 130
remote objects, 135
securing .NET Remoting, 123–160
security with remoting in .NET 2.0, 154
server accepting authenticated requests, 145
unsafeAuthenticatedConnectionSharing attribute, 90
useAuthenticatedConnectionSharing attribute, 90
using a GenericIdentity, 130
Windows authentication, 130, 134
enabling, 137
authentication protocols, 124
Kerberos, 126
NTLM authentication, 124
security package negotiate, 128
Security Support Provider Interface, 128
authenticationLevel property
provider tag, 144
authenticationMode property
.NET Remoting v2.0, 159
security with remoting in .NET 2.0, 151, 154
authorization
see also security
brief description, 123
custom application authorization, 149
GenericPrincipal, 130
implementing authorization in server, 149
IsInRole method, 133

links to web sites, 542
types, .NET Framework, 133
WindowsPrincipal, 130
AutoLog property, 110

B

base objects, 11
Base64 encoding, 439, 443
BaseChannelObjectWithProperties class, 539
BaseChannelSinkWithProperties class, 540
client-side sinks, 361
BaseChannelWithProperties class, 540
default .NET Remoting channels, 462
implementing client-side channels, 446
basic authentication, 134
BeginInvoke method
asynchronous IMessageSink processing, 338
concerns regarding SoapSuds, 287
creating client-side sink and provider, 451
IAsyncResult, 491
Visual Studio 2002 behavior, 52
when to use .NET Remoting, 281
behaviors
changing default remoting behavior, 359
versioning behavior, 95
binary encoding via HTTP, 96
binary formatter
links to web sites, 548
BinaryClientFormatterSink class, 503
BinaryClientFormatterSinkProvider class, 502
BinaryFormatter class
avoiding version mismatch, 409–413
for HTTP channel, switching to, 96
scalable remoting rules, 280
serializing messages through formatters, 329
version incompatibility, 309–311
when to use .NET Remoting, 278
BinaryServerFormatterSink class, 503
server-side messaging, 336
BinaryServerFormatterSinkProvider class, 501
configuring typeFilterLevel in code, 94
binding
strictBinding attribute, 92
versioning behavior, 95
bindingRedirect tag
versioning CAOs, 242
bindTo attribute
channel tag, 90
Break method, 304, 305
breakpoints
debugging Windows service, 115
manual breakpoints, 304
BroadcastEventWrapper class, 217, 219
broadcasting
MSMQ, 284
UDP broadcasts, 282
BroadcastMessage method, 215
BuildFormatName method, 286
business logic
checking parameters in IMessageSink, 482
implementation level, 470
moving constraints to metadata level, 472
scenarios for .NET remoting, 4
sinks, 419
ByRef objects, 13, 26
ByValue objects, 13, 25

C

C# Corner web site, 546
caching
cached credentials, 144
designing applications for static scalability, 299
explicit caches, 301
grouping data as static/dynamic, 300
links to web sites, 548
scalable remoting rules, 281
call value
authenticationLevel property, 144
callbacks
deserialization of object security, 93
encryption, 376
receiving from server through delegates, 94
scalable remoting rules, 281
when to use .NET Remoting, 277, 278
CallContext class, 512
accessing CallContext on server side, 211
accessing directly in code, 212
ILogicalThreadAffinative, 209
LogicalCallContext, 514
OOB (out-of-band) data, 209
scope, 211
security, 213
storing LogSettings in CallContext, 210
transferring runtime information, 209–213

CAO (client-activated objects)
ActivatedServiceTypeEntry, 494
C# code of client application, 84
client-side output using, 37, 85
client-side sinks, 352
creating, 34
using factory design pattern, 38
implementing server-side channels, 455, 457
instantiating server-side CAO, 100
ObjRef objects, 324, 325
registering, 96
scalable remoting rules, 281
server-side output using, 38
server-side sponsors, 207
SoapSuds or Interfaces, conclusion, 287
state, 34
versioning, 240–242
versioning with interfaces, 255
when to use .NET Remoting, 277
CAS (code access security), 129
certificates
encryption and IIS, 139
chained channels, 546
chains
see sinks
Channel Framework Extension
links to web sites, 545
channel sinks
BaseChannelSinkWithProperties, 540
channelSinkProviders tag, 86
encapsulating SMTP/POP3 protocol, 426
extending compression sinks, 371
IChannelSinkBase, 532
IClientChannelSink, 532
IClientChannelSinkProvider, 533
IServerChannelSink, 534
IServerChannelSinkProvider, 535
scalable remoting rules, 281
channel tag, 89–91
channel templates, 86
configuration file using, 76, 82, 138
implementing server-side channels, 453
using the IPC channel, 105
using the TCP channel, 154
ChannelData property, 457, 464, 538
ChannelDataStore object, 455
ChannelName property, 456, 463, 499
ChannelPriority property, 456, 464, 499
channels
BaseChannelObjectWithProperties, 539
BaseChannelSinkWithProperties, 540
BaseChannelWithProperties, 540
channel information for IIS, 118
client-side sinks, 351
CrossContextChannel, 336
DispatchChannelSink, 336
encryption, 375
HTTP channel, 17
HttpChannel, 504
HttpClientChannel, 333, 505
HttpServerChannel, 335, 506
IChannel, 537
IChannelInfo, 498
IChannelSender, 539
IChannelSinkBase, 532
IClientChannelSink, 532
IClientChannelSinkProvider, 533
implementing client-side channels, 445–453
implementing server-side channels, 453–462
interprocess communication channel, 102
IPC channel, 102
IServerChannelSink, 534
IServerChannelSinkProvider, 535
Jabber channel, 545
links to web sites, 544–546
Named Pipe channel sample, 102
registering default channels, 462
SDLChannelSink, 335
security with remoting in .NET 2.0, 151
suppressChannelData attribute, 90
TcpChannel, 506
TcpClientChannel, 507
TcpEx channel, 545
TcpServerChannel, 508
transport channels, 321
wrapping transport channel, 462–465
Channels namespace, 499–504, 531–540
channels tag, 89
ChannelServices, 500
configuration files, 85, 86
using SmtpChannel, 465, 466
Channels.Http namespace, 504–506
Channels.Tcp namespace, 506–508
ChannelServices class, 500
character encoding, 424
channelSinkProviders tag, 85, 86

CheckableContextProperty class, 474
CheckAndStartPolling method, 435
CheckAttribute class, 478
CheckerSink class, 475, 477, 483
chunky interfaces, 278
class library project, 228
classes
  identifying namespace, 186
click event
  creating Windows Forms client, 168
client assembly
  asynchronous calls, 53, 55
  multiserver configuration, 66
  non-wrapped proxy metadata, 72, 73
  one-way calls, 56
  output when removing OneWay attribute, 58
  remoting application architecture, 14
  sample remoting application, 18, 21
  synchronous calls, 49, 50
  wrapped proxies, SoapSuds, 70
Client class
  factory design pattern creating CAO, 41
  sample remoting application, 18
  server-activated objects, 27
client tag, 85, 98
client-activated objects
  see CAO
client-side channels
  client-side sinks, 351
  IClientChannelSinkProvider, 533
  SMTPClientChannel, 445–453
client-side messaging, 331–333
client-side proxies
  RealProxy, 531
client-side sinks, 350–353, 361–364
  changing sinks programming model, 408
  compression sinks, 359, 362, 363
  configuration files, 369
  connection using/not using compared, 370
  creating encryption sinks, 380
  EncryptionClientSink, 380
  IClientChannelSink, 532
  IClientFormatterSink, 534
  IClientFormatterSinkProvider, 534
  implementing client-side channels, 449–453
  passing runtime information, 390, 392
  sink providers, 367
client-side sponsors, 197
  calling expired object's method, 198–203
  scalable remoting rules, 281
  when to use .NET Remoting, 277, 278
client-side transport channel sinks
  encapsulating SMTP/POP3 protocol, 426
client/server affinity
  creating NLB clusters, 296
ClientChannelSinkStack
  handling asynchronous response, 345
clientConnectionLimit attribute
  channel tag, 90, 91
ClientContextTerminatorSink
  client-side messaging, 332
  dynamic sinks, 356
clientProviders tag
  channel tag, 202
  client-side sinks, 400
  formatter tag, 96
  HttpClientChannel, 506
  provider tag, 143
  sink providers, 349
clients
  ActivatedClientTypeEntry, 494
  back-end-based client, 169–184
    ASP.NET based clients, 169
    remoting components hosted in IIS, 172, 176
    security, 177
  BinaryClientFormatterSink, 503
  BinaryClientFormatterSinkProvider, 502
  client for second server component, 175
  client-side sponsors, 196, 197
  configuring server objects in client configuration, 165
  console client, 163–166
  end-user client security, 177
  HttpClientChannel, 505
  lifecycle management, versioned SAOs, 236, 237, 239
  notification of clients, 277
  remoting clients, 161–184
  remoting events, 214
  RemotingClientProxy, 517
  servers requiring different versions, 259, 260, 263, 267
  SoapClientFormatterSink, 503
  SoapClientFormatterSinkProvider, 503
  TcpClientChannel, 507

transfer runtime information with server, 209–213
versioning with interfaces, 249, 251, 253, 255
WellKnownClientTypeEntry, 496
Windows Forms client, 167–169
ClientSponsor class, 510
clipboard sharing
links to web sites, 546
cloning
StreamingContext structure, 520
CLR
locating assemblies, 227
clusters
see also NLB clusters
caching, 299
designing applications for static scalability, 299
IP addresses for, 291
request distribution, 291
scaling out remoting solutions, 290
code access security (CAS), 129
codeBase
CLR resolving assembly references, 227
COM
links to web sites, 543
COM+, 6
command-line tools, 226
commands
SMTP Response code classes, 422
Common Object Request Broker Architecture
see CORBA
communication
interprocess communication channel, 102
Complete method
AsyncResult, 512
compression sinks, 359–375
client-side sink chain, 359
client-side sinks, 362, 363
extending, 371–375
server-side sink chain, 360
CompressionClientSink class, 360, 368
CompressionClientSinkProvider class, 360, 367, 369
CompressionHelper class, 363
CompressionServerSink class, 360, 369
CompressionServerSinkProvider class, 360, 368, 370
CompressionSink assembly, 369
confidentiality, 124
.config extension, 175
configuration classes, 493–497
ActivatedClientTypeEntry, 494
ActivatedServiceTypeEntry, 494
multiserver configuration, 59
RemotingConfiguration, 492–493
TypeEntry, 493
WellKnownClientTypeEntry, 496
WellKnownObjectMode enumeration, 497
WellKnownServiceTypeEntry, 495
configuration files, 76
see also web.config file
application configuration files, 163
BinaryFormatter version mismatch, 412
changing base lifetime example, 188
changing default lease time, 187
changing sinks programming model, 402, 408, 409
client-side sinks, 369
passing runtime information, 400
Configure method, 76
creating encryption sink providers, 386
functionality, 76
general configuration options, 85
implementing client-side channels, 445
main reason for using, 75
naming conventions, 76
problem with SoapSuds, 77
solution, 80
protecting from users, 75
remoting components hosted in IIS as clients, 174
server-side sinks, 370
sink providers, 349
standard configuration options, 85
structure of file, 85
tags
activated tag, 97, 99
application tag, 85
appSettings tag, 165
assemblyBinding tag, 241
bindingRedirect tag, 242
channel tag, 89
channels tag, 86, 89
channelSinkProviders tag, 86
client tag, 98
clientProviders tag, 96
configuration tag, 85
debug tag, 86

formatter tag, 92
lifetime tag, 88
provider tag, 92
service tag, 96
troubleshooting, 305–309
using in application, 76, 82
using SmtpChannel, 465, 466
wrapping transport channel, 462
configuration tag, 85
Configure method
ASP.NET based clients, 171
changing base lifetime example, 189
configuration file settings, 305, 307
creating Windows Forms client, 168
remoting components hosted in IIS as clients, 172, 174
using configuration files, 76
Connect method, 493
connection pooling, 90
connectionGroupName attribute
channel tag, 90
connections
clientConnectionLimit attribute, 90
connectionGroupName attribute, 90
connection using/not using sinks compared, 370, 371
NLB clusters, 292
POP3Connection class, 428
SmtpConnection class, 426, 438, 440
TCP connections, 292
unsafeAuthenticatedConnectionSharing attribute, 90
useAuthenticatedConnectionSharing attribute, 90
console applications
creating console clients, 163
creating server for remoting clients, 161
deploying server application, 108
console client implementation
creating console clients, 163
constraints
moving constraints to metadata level, 471
ConstructionCall class
properties, 327
ConstructionCall messages, 326, 352
ConstructionCallMessage, 240
constructors
IConstructionCallMessage, 530
IConstructionReturnMessage, 530
content reply
POP3 replies, 423
content-type header
BinaryFormatter version mismatch, 409
Context property
StreamingContext structure, 520
ContextAttribute
ContextBoundObject, 472
intercepting calls, 473
ContextBoundObject, 472, 476, 477
contexts
CallContext, 512
CheckableContextProperty, 474
ClientContextTerminatorSink, 332
CrossContextChannel, 336
dynamic sinks, 356
GetPropertiesForNewContext method, 472
IContextProperty, 473
IsContextOK method, 472, 483
IsNewContextOK method, 473, 474
LogicalCallContext, 514
ServerContextTerminatorSink, 337
StreamingContext structure, 520
CopyLocal property
.NET Framework versioning, 229
CORBA (Common Object Request Broker Architecture)
introduction, 5
lifetime management, 185
links to web sites, 544
CreateInstance method, 488
ActivatedClientTypeEntry, 494
configuring server objects in client configuration, 166
IConstructionCallMessage, 530
CreateInstanceFrom method, 488
CreateMessageSink method
client-side sinks, 352
implementing client-side channels, 446, 448
sinks using custom proxy, 414
wrapping transport channel, 463
CreateProxy method
deploying remote applications, 100
ProxyAttribute, 531
CreateServerChannelSinkChain method, 355
CreateServerObjectChain method, 337
CreateSink method
client-side sinks, 352, 353
creating client-side sink and provider, 449, 450

IClientChannelSinkProvider, 533
IServerChannelSinkProvider, 536
passing runtime information, 393, 399
server-side sinks, 355
credentials
cached credentials, 144
useDefaultCredentials attribute, 90
cross-AppDomain remoting, 276
cross-process on multiple machines in LAN, 276
cross-process on single machine, 276
cross-process via WAN/Internet, 278
CrossContextChannel
dynamic sinks, 356
server-side messaging, 336
cryptographic process
encryption helper, 379
CurrentPrincipal property
.NET Remoting v2.0 based server, 156
implementing authorization in server, 149
CurrentState property
LeaseState enumeration, 511
custom attributes, 92
custom channels
links to web sites, 544–546
custom exceptions, 288
troubleshooting using, 313–314
custom marshaling
links to web sites, 543
custom proxies
sinks using, 413–419
Customer class, 15
CustomerManager class, 209
CustomerManager SAO
accessing, 97
configuration files, 82
server startup code, 82
CustomProxy class, 414

D

data
see also metadata
ChannelData property, 457, 464, 538
ChannelDataStore object, 455
GetChannelData method, 536
GetObjectData method, 519
grouping data as static/dynamic, 300
MessageData objects, 323
OOB (out-of-band) data, 209
RemotingData property, 337
SetData method, 210
SinkProviderData objects, 407
suppressChannelData attribute, 90
typed DataSets, 287
User Datagram Protocol (UDP), 277
DATA command, SMTP, 423
data object definition, 15
data serialization, 12
DCE (Distributed Computing Environment), 5
DCOM (Distributed Component Object Model), 5, 185
debug tag, 85, 86, 87, 308
debugging
changing default remoting behavior, 359
configuration file settings, 305–309
IIS, 120, 303
Just-In-Time debugging, 305
manual breakpoints, 304
remoting components hosted in IIS as clients, 174, 175
server applications, 303
troubleshooting hints, 303–305
Windows service, 113
in real runtime state, 114
selecting type of program, 115
setting breakpoints, 115
declarative security, 133
DefaultLifeTimeSingleton, 188, 189, 191, 193
DELE command, POP3, 424, 432
Delegate class, 490–491
delegate value
impersonationLevel property, 143, 147
delegates, 51
asynchronous calls, 51, 52
asynchronous delegates, 434, 441
callbacks, 94
classes and delegates, 490
creating, 52
declaration of, 51
deserialization of object security, 93
IAsyncResult, 491
method signatures, 51
remoting events, 218
one way methods/events, 222
typeFilterLevel changing security, 312
DeleteMessage method, 432
demilitarized zone (DMZ), 4, 317
deployment, 108
using IIS, 116

deserialization of objects, 93
dictionary keys
  corresponding message property, 327
digest authentication, 134
DIME
  links to web sites, 546
Disconnect method, 432
DispatchChannelSink
  server-side messaging, 336
DispatchException method, 346
displayName attribute
  channel tag, 89
  client tag, 98
  wellknown tag, client tag, 99
  wellknown tag, service tag, 97
Dispose method, 517
distributed applications
  development of, 3
  transferring runtime information, 209
  when to use .NET Remoting, 275
Distributed Component Object Model (DCOM), 5, 185
Distributed Computing Environment (DCE), 5
distributed reference counting, 185
distributed transactions, 280, 281
DMZ (demilitarized zone), 4
  troubleshooting client behind firewall, 317
DoCheck method, 479, 480
Donate method, 479, 483
DumpMessageContents method, 417
DumpObjectArray method, 418
dynamic sinks, 356–357
  client-side messaging, 332

E

e-mail
  checking for new mail, 433
  creating e-mail headers, 425
  mapping protocols to .NET Remoting, 438, 441
  Mercury/32 e-mail server, 467
  parsing e-mail address for incoming request, 444
  parsing URL for e-mail address, 444
  POP3 protocol, 433
  protocols transferring e-mail, 422
  SMTP response codes, 422
  types of, 441
EJB (Enterprise Java Beans), 6
encoding
  binary encoding via HTTP, 96
  character encoding, 424
  converting string to Base64 encoding, 439
encryption
  see also security
  .NET Remoting, 375–390
  .NET Remoting v2.0, 157
  asymmetric/symmetric combination, 375
  changing default remoting behavior, 359
  deserialization of object security, 93
  HTTP channel, 375
  HTTPS/SSL, 375
  IIS, 138, 375
  network sniffing encrypted traffic, 159
  network sniffing unencrypted versions, 157
  providers, 386–390
  security with remoting in .NET 2.0, 154
  sinks, 380–385
  SSL encryption, 139
  symmetric encryption, 376–380
  TCP channels, 375
encryption helper
  cryptographic process, 379
  symmetric encryption, 378
encryption property.0, 152
EncryptionClientSink class, 380
EncryptionClientSinkProvider class, 386, 387
EncryptionServerSink class, 383
EncryptionServerSinkProvider class, 388
EndInvoke method, 491, 512
Enterprise Java Beans (EJB), 6
Enterprise Services, 280, 281
EnterpriseServicesHelper class, 516
ERR message, POP3, 423
errors
  Configure method, 88
  debug tag, 86
  impersonation error, 147
  SoapFault, 521
event handling
  ASP.NET based clients, 172
  encryption, 376
  intermediate wrapper, 218
event logs
  porting remoting server to Windows services, 110

EventInitiator
  remoting events, 221
EventLog viewer
  installing Windows service, 112
events
  one way events, 222, 224
  remoting events, 213–224
  scalable remoting rules, 281
  typeFilterLevel changing security, 311
  when to use .NET Remoting, 277, 278, 281
exception classes, 497–498
exceptions
  BinaryFormatter version mismatch, 409–413
  typeFilterLevel changing security, 312
  custom exceptions, 288
  expired TTL, 187
  extending compression sinks, 371
  handling asynchronous response, 346
  remoting events, 217
  RemotingException, 497
    passing runtime information, 394
  RemotingTimeoutException, 497
  SerializationException, 309, 521
  ServerException, 497
  sinks using custom proxy, 415
  versioning serializable objects, 242, 243
expired object's method, 198
ExtendedMBRObject class, 193, 194
  server-side sponsors, 206

F

factory design pattern
  creating CAOs using, 38, 39, 40, 41
factory object
  client-side output using, 42
  server-side output using, 43
filters
  TypeFilterLevel enumeration, 521
  typeFilterLevel attribute, 92
fine-grained security, 280
fingerprints
  strong naming, 225
firewalls, 317
  lifetime management, 185
formatter providers
  channel tag, 91
    client-side sponsors, 202
    optional additional attributes, 92
formatter tag
  clientProviders tag
    BinaryFormatter version mismatch, 412
    changing sinks programming model, 402
    configuration file using attributes, 96
    includeVersions attribute, 95
    sink providers, 350
  IClientFormatterSink, 534
  IClientFormatterSinkProvider, 534
  serverProviders tag
    attributes, 92
    strictBinding attribute, 95
    configuration file using attributes, 91, 94, 96
  typeFilterLevel attribute, 92
formatters
  binary formatter, 548
  BinaryClientFormatterSink, 503
  BinaryClientFormatterSinkProvider, 502
  BinaryFormatter, 96, 278, 280
    avoiding version mismatch, 409–413
    serializing messages through formatters, 329
    version incompatibility, 309–311
  BinaryServerFormatterSink, 336, 503
  BinaryServerFormatterSinkProvider, 501
  brief description, 321
  IClientFormatterSink, 328, 534
  IClientFormatterSinkProvider, 534
  messages, 328
  serializing message objects through, 329–330
  SOAP formatter, 504, 505, 548
  SoapClientFormatterSink, 333, 503
  SoapClientFormatterSinkProvider, 503
  SoapFormatter, 329
  SoapServerFormatterSink, 336, 504
  SoapServerFormatterSinkProvider, 502
Formatters namespace, 521–523
FormsIdentity, 130
FreeNamedDataSlot method, 212
Freeze method, 473, 474
full assembly name, 225
Full value, typeFilterLevel attribute, 93

G

GAC (Global Assembly Cache), 89, 233, 227–229
gacutil.exe, 228, 234, 235, 238
garbage collection, 185

general assembly
see also shared assemblies
remoting application, 14, 20
server-activated objects, 26
general.dll assembly, 47
one-way calls, 56
remoting events, 214
versioning with interfaces, 253
generated metadata assembly, 12
GenericIdentity, 130
GenericPrincipal, 130
Genuine Channels, 279
GetAuthenticationEntry method, 403
GetChannel method, 89
GetChannelData method, 536
GetChannelSinkProperties method, 135
GetCleanAddress method, 445, 460
GetCompressedStreamCopy method
client-side sinks, 363, 364
extending compression sinks, 374
server-side sinks, 367
GetCurrent method, 130
GetDynamicSink method, 356
GetEnumerator method, 537
GetExceptionIfNecessary method, 410, 412
GetLeaseInitial method, 186
GetLifetimeService method, 197
GetMessage method, 431
GetObject method, 488
Connect method, 493
creating proxies, 322
deploying remote applications, 100
metadata, configuration files, 77
GetObjectData method
custom exceptions, 288, 290
ISerializable, 519
servers requiring different versions, 266
StreamingContext structure, 520
versioning serializable objects, 243, 245
GetPerson method, 258, 261, 268, 271
GetPropertiesForNewContext method, 472
GetRegisteredWellKnownClientTypes method, 101
GetRequestStream method, 533
GetResponseStream method
creating server-side sinks, 458
IServerChannelSink, 334
server-side sinks, 365
getSAOVersion method, 236
GetTransparentProxy method, 415
GetUncompressedStreamCopy method
client-side sinks, 363, 364
extending compression sinks, 372, 373
server-side sinks, 366
GetURLBase method, 456
GetUrlsForUri method
IChannelReceiver, 538
implementing server-side channels, 457
wrapping transport channel, 464
Global Assembly Cache
see GAC
Global.asax file, 171
granularity, 280
groups, 149
connectionGroupName attribute, 90

H

HandleAsyncResponsePop3Msg method, 441, 442, 453
HandleIncomingMessage method, 441, 442, 459, 460
HandleMessage method, 219
HandleMessageDelegate method, 434
HandleReturnMessage method, 323
Hashtables, SMTPHelper class, 437
headers
creating e-mail headers, 425
ITransportHeaders, 536
HELO command, SMTP, 423
helper classes
EnterpriseServicesHelper, 516
SoapSuds or Interfaces, conclusion, 287
hooking
listen attribute, 90
HTTP channel
see also channel tag
binary encoding via HTTP, 96
client for second server component, 176
configuration information, 89
deployment using IIS, 116
encryption, 375
multiserver configuration, 64
referencing predefined channel, 89
remoting components hosted in IIS as clients, 174
sample remoting application, 17, 19
scalability features, 293
security related properties, 90
switching to BinaryFormatter for, 96
HTTP Content-Length header, 371
HTTP KeepAlives, 280

Http namespace, 504–506
HTTP proxies, 279, 297
HTTP requests
  extending compression sinks, 375
  TCP connections, 292
HTTP responses
  extending compression sinks, 375
HttpApplication class, 171
HttpChannel class, 504
  registering default .NET Remoting channels, 462
  scalable remoting rules, 280
  when to use .NET Remoting, 278
HttpClientChannel class, 505
  client-side messaging, 333
  client-side sinks, 352
  IChannelSender, 539
  registering, 462
HttpServerChannel class, 506
  registering, 462
  server-side messaging, 333, 335
  server-side sinks, 354
HttpServerSocketHandler class, 333
HttpServerTransportSink class, 335

I

/i parameter, 228, 234
IAsyncResult interface, 491, 512
IBroadcaster interface, 214, 215
IChannel interface, 499, 537
  client-side channels, 446, 447
  client-side sinks, 351
  server-side channels, 454
IChannelInfo interface, 498
IChannelReceiver interface, 454
IChannelSender interface, 446, 448, 539
IChannelSinkBase interface, 458, 532
IClientChannelSink interface, 328, 329, 360, 532
  asynchronous messaging, 340–342
  changing sinks programming model, 409
  client-side sink providers, 368
  client-side sinks, 361
  creating client-side sink and provider, 452
  interfaces for message sinks, 328
  passing runtime information, 392, 396
  sinks using custom proxy, 419
IClientChannelSinkProvider interface, 360, 533
  client-side sink providers, 368
  creating client-side sink and provider, 449
  implementing client-side channels, 446
  passing runtime information, 392
IClientFormatterSink interface, 534
  formatters, 328
IClientFormatterSinkProvider interface, 534
IClientResponseChannelSinkStack interface, 341
IConstructionCallMessage interface, 530
IConstructionReturnMessage interface, 530
IContextProperty interface, 473
IContributeDynamicSink interface, 356
IContributeObjectSink interface, 474
ICustomerManager interface, 14, 15, 16, 18, 20
identify value
  impersonationLevel property, 143
identities, 129, 130
Identity objects, 323, 325
IDictionary interface
  client-side channels, 446
  server-side sinks, 354
IDynamicMessageSink interface, 356
IDynamicProperty interface, 356, 357
IEnvoyInfo interface, 498
IIdentity interface, 129
IIS (Internet Information Server)
  ASP.NET application impersonating client, 183
  ASP.NET based clients, 171
  authentication with IIS, 133, 134, 135
  client for second server component, 176
  configuration file debugging, 308
  debugging, 120
  debugging hints, 303
  deployment for anonymous use, 118
  deployment using, 116
  encryption, 138, 375
  end-user client for IIS hosted component, 177
  intermediary Remoting server hosted in, 173
  membership in Windows groups, 149
  output window for IIS hosted server, 177
  preparing to use IIS as container, 117
  remoting components hosted as clients in, 172, 176
  scalable remoting rules, 280
  security without IIS, 140
  server-side objects, 116
  troubleshooting using custom exceptions, 313

ILease interface, 186, 508
client-side sponsors, 197
CurrentState property, 511
Register method, 197
ILogicalThreadAffinative interface, 209, 514
IMessage interface, 525
how messages work, 326
passing runtime information, 390
IMessageSink interface, 328, 329, 526
asynchronous messaging, 338–339
AsyncProcessMessage method, 328
client-side sinks, 353
intercepting calls, 469
interfaces for message sinks, 328
NextSink property, 328
passing runtime information, 390, 392
proxies creating messages, 323
server-side asynchronous processing, 347
sinks using custom proxy, 419
SyncProcessMessage method, 328
IMethodCallMessage interface, 528, 529
IMethodMessage interface, 527
IMethodResponseMessage interface, 529
IMethodReturnMessage interface, 528
imperative security checks, 133
impersonate value
impersonationLevel property, 143, 147
impersonation, 182
.NET 1.x and 2.0 differences, 152
impersonation error, 147
impersonationLevel property
.NET Remoting v2.0, 159
provider tag, 143
security with remoting in .NET 2.0, 151
server accepting authenticated requests, 147, 148
implementation of .NET Remoting
advantages of .NET Remoting, 9
client implementation, 18
server implementation, 15
In-Reply-To header
creating e-mail headers, 425
mapping protocols to .NET Remoting, 441
includeVersions attribute
BinaryServerFormatterSinkProvider, 501
formatter tag, clientProviders tag, 95
formatter tag, serverProviders tag, 92
servers requiring different versions, 264
versioning serializable objects, 242
Indigo, 257
InfinitelyLivingSingleton, 188, 189, 191, 193
InfinitelyLivingSingleton_LifeTime property, 195, 196
Ingo's .NET Remoting FAQ corner, 541
InitFields method, 323
initialization vector (IV)
symmetric encryption, 378
InitializeLifetimeService method
changing base lifetime, 188, 189, 191, 193
changing lease time, 187, 188
ExtendedMBRObject, 193, 194
ILease, 186, 508
lifetime management, 43
server overriding, 46
InitialLeaseTime property, 186
InitTypeCache method, 100
installutil.exe, 109, 112, 113
instances, .NET Framework, 30
InstanceSponsor_Lifetime, 205
InstanceSponsor_RenewOnCallTime, 205
integrated security, 138
integrity, 124
interception
CheckerSink, 477
ContextAttribute, 473
IMessageSinks, 469
Organization, 476
interface definitions
client and server accessing DDL, 14
generated metadata assembly, 12
multiserver configuration, 62
sample remoting application, 14
interfaces
advantages of .NET Remoting, 11
chunky interfaces, 278
client and server accessing, 14
configuring server objects in client configuration, 165
deploying remote applications, 100
extending .NET Remoting framework, 525
IChannel, 499, 537
IChannelInfo, 498
IChannelSender, 539
IChannelSinkBase, 532
IClientChannelSink, 532
IClientChannelSinkProvider, 533
IClientFormatterSink, 534
IClientFormatterSinkProvider, 534

IConstructionCallMessage, 530
IConstructionReturnMessage, 530
IEnvoyInfo, 498
ILease, 508
IMessage, 525
IMessageSink, 526
IMethodCallMessage, 528
IMethodMessage, 527
IMethodResponseMessage interface, 529
IMethodReturnMessage, 528
IObjectHandle, 498
IRemotedType, 103
IRemotingTypeInfo, 498
ISerializable, 519
IServerChannelSink, 534
IServerChannelSinkProvider, 535
ISponsor, 509
ITrackingHandler, 517
ITransportHeaders, 536
message sinks, 328
servers requiring different versions, 258
shared assembly defining, 102
shared interfaces, 11, 39
SoapSuds or Interfaces, 286, 287
versioning with, 246–256
interoperability, 544
interprocess communication, 102
Intrinsyc's Ja.NET, 547
Invoke method, proxies, 416, 419
IObjectHandle interface, 498
IP addresses
useIpAddress attribute, 90
IPC channel, 102, 105, 106
IPrincipal interface, 129
IRemoteCustomerManager interface, 100
IRemotedType interface, 103
IRemoteFactory interface, 161, 198
IRemoteObject interface, 63, 198
IRemoteSecond interface, 161, 173
IRemotingTypeInfo interface, 498
IsContextOK method, 472, 483
ISerializable interface, 12, 519
deserialization of object security, 93
versioning serializable objects, 243, 244
IServerChannelSink interface, 360, 534
creating server-side sinks, 458
interfaces for message sinks, 328
server-side asynchronous processing, 347
server-side messaging, 334
server-side sink providers, 368
server-side sinks, 364
sinks using custom proxy, 419
IServerChannelSinkProvider interface, 360, 535
passing runtime information, 398
server-side sink providers, 369
IsInRole method, 133
IsNewContextOK method, 473, 474
ISponsor interface, 196, 200, 509
isServer attribute
wrapping transport channel, 462
ITrackingHandler interface, 517
links to web sites, 547
ITransportHeaders interface, 536
extending compression sinks, 371
mapping protocols to .NET Remoting, 438
moving messages through transport channels, 330
IUSR_MACHINENAME
anonymous access, 134
IWorkerObject interface, 63

J

Ja.NET, 547
Jabber channel, 545
Java RMI (Java Remote Method Invocation)
introduction, 6
lifetime management, 185
links to web sites, 544
Just-In-Time debugging, 305

K

KDC (Key Distribution Center)
Kerberos authentication, 126
KeepAlive method, 204, 206, 208
KeepAlives
disabling, 293
scalable remoting rules, 280
TCP connections, 292
Kerberos, 126
MSDN security samples, 144
key pairs, 226
.NET Framework versioning, 228
versioning with interfaces, 247
KeyGenerator application
symmetric encryption, 376, 378

L

/l parameter, 228, 235, 238
language agnostic names, 149
layers, 4

Lease class, 186
LeaseManager class
  ISponsor, 196
  leaseManagerPollTime attribute, 187
  LeaseTimeAnalyzer method, 186
  lifetime management, 186
  server-side sponsors, 204
  sponsors, 196
  valid units of measurement, 187
leaseManagerPollTime attribute
  LeaseManager, 187
  lifetime tag, 88, 188
leases, 186, 187, 508
LeaseSink, 337
LeaseState enumeration, 511
leaseTime attribute
  lifetime tag, 88, 188
LeaseTimeAnalyzer method, 186
lifecycle management
  versioned SAOs, 233, 234–239
  versioning CAOs, 240
lifetime management
  see also time
  advantages of .NET Remoting, 12
  changing base lifetime, 188, 189, 191, 193
  client calling timed-out CAO, 45, 46
  DefaultLifeTimeSingleton, 188
  distributed reference counting, 185
  garbage collection, 185
  GetLifetimeService method, 197
  InfinitelyLivingSingleton_LifeTime property, 195
  InitializeLifetimeService method, 186
  InstanceSponsor_Lifetime, 205
  LeaseManager, 186
  leases, 186
  Lifetime namespace, 508–511
  Lifetime property, 195, 205
  lifetime tag, 85, 88
    changing base lifetime, 188
  LifetimeServices class, 511
  managing object lifetime, 185
  nondefault lifetimes, 193, 194
  server-side sponsors, 203
  Singleton objects, 30
  sponsors, 43, 196–209
  sponsorship, 185
  TimeSpan properties, 186
  TTL (time-to-live), 185
links to web sites, 541–548
LIST command, POP3, 424, 428
listen attribute, channel tag, 90
listeners
  remoting events, 214
loadTypes attribute, debug tag, 87
local storage, 299
LocallyHandleMessageArrived method, 218
logging, 209, 210
LogicalCallContext class, 514
LogicalCallContext property, 327
  passing runtime information, 391, 393, 397
logon process, 137, 138
LongerLivingSingleton class, 188, 189, 191, 193
  properties, 195, 196
Low value, typeFilterLevel attribute, 93

M

machineName attribute, 90, 317
mail
  see e-mail
MAIL FROM command, SMTP, 423
maintenance, 292
major version, 225
managed extensions, 543
MapPath method, 172
mapping protocols to .NET Remoting, 437–445
marshal by value object, 489
MarshalByRefObject class, 488–489
  changing base lifetime, 188, 189, 191, 193
  changing lease time, 187
  deployment using IIS, 116
  ExtendedMBRObject, 193
  leases, 186
  nondefault lifetimes, 193, 194
  objects, 26, 34, 59, 93
  remoting events, 215
  sample remoting application, 13, 14, 16
  scalable remoting rules, 281
  server-side sponsors, 203
  sponsors, 185
  troubleshooting multihomed machines, 315, 316
  versioning with interfaces, 256
marshaling, 543
Mercury/32 e-mail server, 467
message confidentiality, 124
message objects
  passing runtime information, 390
  proxies, 322, 323
  serialization through formatters, 329–330

message queuing, 277
message sinks, 328–329
  brief description, 321
  creating proxies, 323
  how messages work, 326
Message-Id header, 425, 441
MessageArrived event, 215, 222
MessageArrivedLocally event, 218, 219
MessageCount property, POP3, 430
MessageData objects, 323
MessageReceived method, 434, 441
messages, 326–331
  Args property, 327
  AsyncProcessMessage method, 328
  asynchronous messaging, 338–348
  brief description, 321
  BroadcastMessage method, 215
  client-side messaging, 331–333
  ConstructionCall messages, 326, 352
  ConstructionCallMessage, 240
  contents described, 327
  CreateMessageSink method, 352
  DeleteMessage method, 432
  DumpMessageContents method, 417
  formatters, 328
  GetMessage method, 431
  HandleIncomingMessage method, 441
  HandleMessage method, 219
  HandleMessageDelegate method, 434
  HandleReturnMessage method, 323
  how they work, 326
  IConstructionCallMessage, 530
  IConstructionReturnMessage, 530
  IDynamicMessageSink, 356
  IMessage, 525
  IMessageSink, 338–339, 526
  IMethodCallMessage, 528
  IMethodMessage, 527
  IMethodResponseMessage interface, 529
  IMethodReturnMessage, 528
  LocallyHandleMessageArrived method, 218
  MessageArrived event, 215, 222
  MessageArrivedLocally event, 218, 219
  MessageCount property, POP3, 430
  MessageData objects, 323
  MessageReceived method, 434, 441
  MethodName property, 327
  MethodSignature property, 327
  ProcessMessage method, 333
  ProcessMessageFinish method, 356
  ProcessMessageStart method, 356
  properties and dictionary keys, 327
  proxies creating, 323
  SendMessage method, 440
  SendReplyMessage method, 440
  SendRequestMessage method, 440
  SendResponseMessage method, 440
  SerializeSoapMessage method, 330
  serializing messages through formatters, 329
  server-side messaging, 333–338
  sink processing, 531
  SoapMessage, 521
  SyncProcessMessage method, 323
  transport channels, 326
    messages moving through, 330–331
  TypeName property, 327
  Uri property, 327
  WaitAndGetResponseMessage method, 441
Messaging namespace, 512–514
  extending .NET Remoting framework, 525–529
metadata
  .NET Framework versioning example, 232
  .NET Remoting configuration files, 77
  CallContext, 512
  client-side sinks, 400
  CLR locating assemblies, 227
  generated metadata assembly, 12
  ITransportHeaders, 536
  lifecycle management, versioned SAOs, 236, 239
  moving constraints to, 471, 472
  non-wrapped proxy metadata, 72
  shared assemblies, SoapSuds, 68
Metadata namespace, 514–516
method signatures
  delegates, 51
MethodCall class, 326, 327, 529
MethodName property, 327
MethodResponse class, 529
methods
  IMethodCallMessage, 528, 529
  IMethodMessage, 527
  IMethodResponseMessage interface, 529
  IMethodReturnMessage, 528
  MethodCall class, 327, 529
  MethodName property, 327

MethodResponse class, 529
MethodSignature property, 327
one way methods, 222, 224
SoapMethodAttribute, 516
ways of executing in .NET, 46
asynchronous calls, 51
ByValue objects, 25
one-way calls, 55
synchronous calls, 47
MethodSignature property, messages
dictionary key, data type & sample value, 327
Microsoft Management Console
installing Windows service, 111–112
Microsoft Transaction Server (MTS), 6
minor version, 225
mobile objects, 3
mode attribute
wellknown tag, service tag, 82, 97
MSDN
links to web sites, 541–543
MSDN security samples, 140
authentication level client and server, 144
capabilities of security token, 143
message protection, 144
security sample client, 141
server accepting authenticated requests, 145
shared assemblies for .NET remoting, 140
specifying security protocol, 143
MSMQ (Microsoft Message Queue)
asynchronous communication, 421
links to web sites, 545
when to use .NET Remoting, 277, 283
MTS (Microsoft Transaction Server), 6
multicasting
MSMQ, 284
MulticastDelegate, 220
UDP broadcasts, 282
multihomed machines
troubleshooting, 315–317
multiserver configuration, 59
client assembly, 66
examining, 60
HTTP channel, 64
multiserver/multiclient, 13
ports, 64
server assembly, 63, 64, 66
shared assembly, 62
UML diagram, 60

N

name attribute, channel tag, 89
name parameter
SMTPClientChannel, 446
SMTPServerChannel, 454
Named Pipe channel sample, 102
named pipes, 421, 545
names
BuildFormatName method, 286
ChannelName property, 456, 463, 499
connectionGroupName attribute, 90
displayName attribute, 89, 97, 99
FreeNamedDataSlot method, 212
machineName attribute, 90
MethodName property, 327
proxyName attribute, 90
TypeName property, messages, 327
namespaces
see System.Runtime namespaces
naming conventions
configuration files, 76
language agnostic names, 149
strong naming, 225
strongly named assemblies, 226
.NET Framework
authorization check types, 133
concerns regarding SoapSuds, 287
identities, 129
instance creation, 30
principals, 129
shared assemblies, 67
versioning, 225–233
Whidbey, 151
.NET Remoting
advantages of, 9
architecture, 10, 322
client for second server component, 175
configuration files
see configuration files
creating remoting clients, 161
cross-AppDomain remoting, 276
cross-process on multiple machines in LAN, 276
cross-process on single machine, 276
cross-process via WAN/Internet, 278
deployment, 108
using IIS, 116
Enterprise Services or, 280
evolution of remoting, 4
implementation, 9
integrating in Windows services, 108
interface definitions, 11

introduction, 7
lifetime management, 12, 43
mapping protocol to, 440, 437–445
MSDN security samples, 140, 141
multiserver/multiclient, 13
object types, 13
reasons for changing default remoting behavior, 359
remoting components hosted in IIS as clients, 172, 176
remoting events
see under events
sample .NET Remoting application, 13–22
client assembly, 18, 21
client implementation, 18
data object definition, 15
general assembly, 14, 20
interface definitions, 14
server assembly, 15, 21
server implementation, 15
scalable remoting rules, 280
scaling out remoting solutions, 290
scenarios for .NET remoting, 3
security
see security
SoapSuds vs. Interfaces, 286, 287
transferring runtime information, 209
types of remoting, 25, 26
using the IPC channel, 102
version 2.0 based client, 156
version 2.0 based server, 156
versioning, 233–245
ways to run remote objects, 75
web.config file and client configuration, 170
when to use .NET Remoting, 275
.NET Remoting framework
interfaces for extending, 525
links to web sites, 542
nondefault lifetimes, 193, 194
transport channels, 421
.NET Remoting links, 541–548
custom channels, 544–546
Ingo's .NET Remoting FAQ corner, 541
interoperability, 544
MSDN, 541–543
.NET Remoting namespaces
see under System.Runtime namespaces
network latency, 278
Network Load Balancing
see NLB
network traffic
encryption, 375
new operator, 495
IConstructionCallMessage, 530
Next property, 536
NextChannelSink property, 535
NextSink property, 328, 527
NLB (Network Load Balancing)
clusters, 291
nodes, 292
performance, 291
scalable remoting rules, 280
scaling out remoting solutions, 290
when to use .NET Remoting, 277
NLB clusters
connections, 292
creating, 293–299
taking nodes online/offline, 299
words of caution, 294
NLB Manager
creating NLB clusters, 294
nodes
creating NLB clusters, 295, 298
designing applications for static scalability, 299
Network Load Balancing, 292
taking nodes online/offline, 299
NONCE
NTLM authentication, 125
NonSerialized attribute, 489
notification of events, 281
notifications
clients located in same IP subnet, 282
guaranteed asynchronous delivery of, 283
MSMQ, 284
other approaches, 286
when to use .NET Remoting, 282
NT LAN manager (NTLM) authentication, 124
MSDN security samples, 144

O

object pooling, 547
object types, 13
ObjectHandle class, 492
objectUri attribute
wellknown tag, service tag, 82, 97
ObjRef class, 491–492
MarshalByRefObjects object type, 13
properties, 325
proxies, 324
remoting with MarshalByRefObject, 59

observer pattern, 546
OK message, POP3, 423
one way methods/events, 222, 224
one-way calls, 55, 56
  ways of executing methods, 46
OneWay attribute, 57, 58
OneWayAttribute class, 514
OnStart method, 108
OnStop method, 108
OOB (out-of-band) data, 209
Organization class, 470, 476, 479, 483
output caching, 548

P

packetIntegrity value
  authenticationLevel property, 144
packetPrivacy value
  authenticationLevel property, 144
parameters
  checking in an IMessageSink, 480
  i parameter, 228, 234
  l parameter, 228, 235, 238
  name parameter, 446, 454
  pop3Xyz parameters, 446, 454
  PropagateOutParameters method, 323
  replySink parameter, 338, 339
  senderEmail parameter, 446, 454
  smtpServer parameter, 446, 454
  SoapParameterAttribute, 516
  u parameter, 228
Parse method, 499
  implementing client-side channels, 447, 448
  implementing server-side channels, 456
parseURL method, 450
passing by reference, 13, 26
passing by value, 13
  see also serializable objects
PassportIdentity, 130
patterns
  factory design pattern, 38
  observer pattern, 546
per-host/object authentication model, 359
performance
  designing applications for static scalability, 299
  links to web sites, 541
  NLB clusters, 291
persistence
  links to web sites, 547
  StreamingContext structure, 520
Person class, 257, 258, 260, 261, 263, 264, 266, 268, 270, 273
platform independence, 281
platforms, 4
pluggable sink architecture, 402
Poll method, 434
polling
  checking for new mail, 433
  registering POP3 server, 435
pooling, 547
POP3 protocol, 423–424
  asynchronous communication, 421
  checking for new mail, 433
  encapsulating, 427
  POP3 replies, 423
  registering POP3 server, 435
  transferring e-mail, 422
POP3Connection class, 428
POP3Msg class, 427, 443
pop3Password parameter
  SMTPClientChannel, 446
  SMTPServerChannel, 454
POP3Polling class, 433
pop3PollInterval parameter
  SMTPClientChannel, 446
  SMTPServerChannel, 454
POP3PollManager class, 436, 454
pop3Server parameter
  SMTPClientChannel, 446
  SMTPServerChannel, 454
pop3User parameter
  SMTPClientChannel, 446
  SMTPServerChannel, 454
port attribute, channel tag, 89, 200
ports
  channel information for IIS, 118
  configuration files, 82
  creating NLB clusters, 295
  multiserver configuration, 64
  proxyPort attribute, 90
principals
  .NET Framework, 129, 130
  application using, 130
  example with role permission check, 131
  role-based security, 129
priority attribute, channel tag, 89
PriorityChangerSink class, 396
PriorityChangerSinkProvider class, 398
PriorityEmitterSink class, 393
PriorityEmitterSinkProvider class, 395
private assemblies
  shared assemblies compared, 227

private queues, 284
PrivateInvoke method
  proxies creating messages, 323
  proxies returning values, 324
probing
  CLR resolving assembly references, 227
Processes dialog box
  debugging Windows service, 114
ProcessInboundStream method
  creating encryption sinks, 381, 382, 384
  encryption helper, 380
  symmetric encryption, 378
ProcessMessage method
  BinaryFormatter version mismatch, 411
  changing sinks programming model, 406
  client-side messaging, 333
  client-side sinks, 363, 451
  compression sinks, 371, 373
  encryption sinks, 383
  IClientChannelSink, 533
  IServerChannelSink, 334, 535
  mapping protocols to .NET Remoting, 443
  moving messages through transport channels, 330
  passing runtime information, 391, 397
  server-side asynchronous processing, 347
  server-side messaging, 334
  server-side sinks, 366, 458, 459
ProcessMessageFinish method, 356
ProcessMessageStart method, 356
ProcessOutboundStream method
  creating encryption sinks, 381, 382, 384, 385
  encryption helper, 379
  symmetric encryption, 378
PropagateOutParameters method, 323
protocols, 421–425
  authentication protocols, 124, 128
  encapsulating, 426–445
  Kerberos, 126
  mapping to .NET Remoting, 437–445
  POP3 protocol, 423–424
  protocols transferring e-mail, 422
  SMTP protocol, 422–423
  SOAP, 7
  specifying MSDN security protocol, 143
  transport channels, 421
  User Datagram Protocol (UDP), 277
provider tag
  clientProviders tag
    authenticationLevel property, 144
    BinaryFormatter version mismatch, 412
    changing sinks programming model, 402, 408
    client-side sinks, 400
    configuration file using attributes, 142
    impersonationLevel property, 143
    securityPackage property, 143
  serverProviders tag, 91, 92, 146
providers
  see also sink providers
  BinaryServerFormatterSinkProvider, 94
  changing default remoting behavior, 359
  client-side sinks, 352
  CompressionClientSinkProvider, 360, 367, 369
  CompressionServerSinkProvider, 360, 368, 370
  encryption, 386–390
  EncryptionClientSinkProvider, 386, 387
  EncryptionServerSinkProvider, 388
  formatter providers, 91, 92, 202
  IClientChannelSinkProvider, 360, 533
  IClientFormatterSinkProvider, 534
  IServerChannelSinkProvider, 360, 535
  PriorityChangerSinkProvider, 398
  PriorityEmitterSinkProvider, 395
  sink chains, 350
  sink providers, 91
  SinkProviderData objects, 86, 407, 534
  SMTPClientTransportSinkProvider, 449
  SoapClientFormatterSinkProvider, 503
  SoapServerFormatterSinkProvider, 94, 502
  SSPI, 128
  UrlAuthenticationSinkProvider, 86, 407
proxies, 322–325
  brief description, 321
  client-side proxies, 531
  creating messages, 323
  creating proxies, 322
  creating Windows Forms client, 168
  CreateProxy method, 100, 531
  custom proxies, 413–419
  CustomProxy class, 414
  disconnecting an object from, 517
  GetTransparentProxy method, 415
  how they work, 322
  HTTP proxies, 279, 297

lifetime management, 185
non-wrapped proxy metadata, 72
ObjRef, 324
RealProxy, 531
RemotingClientProxy, 517
RemotingProxy objects, 322, 413
returning values, 324
ServerProxy property, 169
sinks using custom proxy, 413–419
SoapSuds generated nonwrapped proxy's source, 240
TransparentProxy objects, 322
when to use .NET Remoting, 279
wrapped proxies, 68, 71
Proxies namespace, 530–531
ProxyAttribute class, 531
proxyName attribute, channel tag, 90
proxyPort attribute, channel tag, 90
published objects, 32, 33
publisher policies
CLR resolving assembly references, 227

Q

queues
private queues, 284
QUIT command, 432, 423, 424

R

RBS (role-based security), 129
RCPT TO command, SMTP, 423
RealProxy class, 531
client-side sinks, 353
creating proxies, 322
proxies creating messages, 323
proxies returning values, 324
sinks using custom proxy, 413
ref attribute
channel tag, 89
formatter tag, 92
provider tag, 92
refactoring
remoting event handling, 217
reference
distributed reference counting, 185
passing by reference, 13, 26
references to web sites, 541–548
Register method, 197
RegisterActivatedServiceType method, 494
RegisterAsyncResponseHandler method, 441
RegisterChannel method, 415
registered sponsors
deserialization of object security, 93
RegisterPolling method
implementing client-side channels, 446, 447
registering POP3 server, 435, 436
RegisterServer method, 444
rejectRemoteRequests attribute, channel tag, 91
cross-process on single machine, 276
Release method
lifetime management, 185
remote components
configuration file debugging, 307
Windows services hosting, 110
remote objects
accessing, 491
authentication, 135
brief description, 3
client-side sponsors, 197
MarshalByRefObject, 489
registering, 99
rejectRemoteRequests attribute, 91
sponsors, 196
Remote Procedure Calls (RPC), 5
remote sponsors, 203
remote users
implementing authorization in server, 149
remoting
see .NET Remoting
Remoting Channel Framework Extension
links to web sites, 545
remoting events, 213–224
one way methods/events, 222, 224
refactoring event handling, 217
scalability, 213
Remoting Management Console (RMC)
links to web sites, 548
Remoting namespaces
see under System.Runtime namespaces
Remoting Probe tool
links to web sites, 548
remoting server
porting to Windows services, 110
RemotingClientProxy class, 517
RemotingConfiguration class, 492–493
Configure method, 76
RemotingData property, 337
RemotingException class, 497
passing runtime information, 394

RemotingHelper class, 100
  classes used for configuration, 494
  client-side sponsors, 198
  configuring IRemoteCustomerManager, 101
  creating console clients, 164
  creating server for remoting clients, 161
  server-side objects for IIS, 116
RemotingProxy objects
  creating proxies, 322
  sinks using custom proxy, 413
RemotingServices class, 493
RemotingTimeoutException class, 497
Renew method, 204
Renewal method, 209
RenewalTime property, 510
RenewOnCall method, 337
renewOnCallTime attribute, lifetime tag, 88, 188
RenewOnCallTime property, 195
  server-side sponsors, 205
  TimeSpan, 186
replySink parameter
  asynchronous IMessageSink processing, 338, 339
request distribution
  clusters, 291
request-for-comment (RFC) documents, 422
requests
  AsyncProcessRequest method, 340
  asynchronous messaging generating, 342–344
  CORBA, 5
  creating server-side sinks, 459
  GetRequestStream method, 533
  HTTP requests, 292
  implementing server-side channels, 455, 457
  mapping protocols to .NET Remoting, 440
  NLB clusters, 292
  parsing e-mail address for incoming request, 444
  rejectRemoteRequests attribute, 91
  RequestSent method, 437
  SendRequestMessage method, 451
  ServiceRequest method, 335
  sinks using custom proxy, 416
RequestSent method
  mapping protocols to .NET Remoting, 440
  registering POP3 server, 437
response codes, 422
ResponseReceived method
  mapping protocols to .NET Remoting, 441, 442
  registering POP3 server, 437
responses
  AsyncProcessResponse method, 340
  AsyncResponseHandler class, 451
  asynchronous messaging handling, 345, 346
  creating e-mail headers, 425
  GetResponseStream method, 334
  HandleAsyncResponsePop3Msg method, 441
  HTTP responses, 375
  IClientResponseChannelSinkStack, 341
  IMethodResponseMessage, 529
  mapping protocols to .NET Remoting, 440
  MethodResponse, 529
  RegisterAsyncResponseHandler method, 441
  SendResponseMessage method, 440
  sinks using custom proxy, 416
  WaitAndGetResponseMessage method, 441
responses hashtable, 437, 438, 440, 441
RETR command, POP3, 424, 431
return values
  proxies, 324
RFC (request-for-comment) documents, 422
RMI (Remote Method Invocation)
  see Java RMI
role-based security (RBS), 129
root
  virtual root, 117
RPC (Remote Procedure Calls), 5
RunInstallerAttribute, 110
runtime information
  passing, 390–401
  transferring client with server, 209–213

S

SAO (server-activated objects), 26
  client-side sinks, 352
  configuration files, 82
  lifecycle management, 233, 234–239
  ObjRef objects, 325
  published objects, 32
  registering, 96, 99
  scalable remoting rules, 280

server-side objects for IIS, 117
server-side output using, 85
SingleCall SAOs, 28, 277
Singleton objects, 30
SoapSuds or Interfaces, conclusion, 287
versioning, 233–239
versioning with interfaces, 246
scalability
designing applications for static scalability, 299
HTTP channel, 293
remoting events, 213
scalable remoting rules, 280
scaling out remoting solutions, 290
when to use .NET Remoting, 275
scope
CallContext, 211
SDLChannelSink
server-side messaging, 335
security, 123–160
see also authentication; authorization; encryption
Access Control Lists, 75
ASP.NET application impersonating client, 182
authentication with IIS, 133
back-end-based client security, 177
CallContext, 213, 512
typeFilterLevel changing security, 311–313
client security token
differences .NET 1.x and 2.0, 152
creating back-end-based client, 177
declarative security, 133
demilitarized zone, 4
deployment using IIS, 116
deserialization of objects, 93
encryption and IIS, 138
end-user client security, 177
fingerprints, 225
HTTP channel attributes, 90
imperative security checks, 133
impersonation
differences .NET 1.x and 2.0, 152
integrated security, 138
key pairs, 226
links to web sites, 541, 543
major concepts, 123
MSDN security, 140, 143
new features also for v2.0, 140
physical separation of layers, 4
protecting configuration files from users, 75
scalable remoting rules, 281
security with remoting in .NET 2.0, 151, 155
servers requiring different versions, 273
transferring runtime information, 209
trusted subsystem, 178
Windows Forms client, 179, 181
without IIS, 140
security mechanisms
code access security (CAS), 129
role-based security (RBS), 129, 133
security package negotiate (SPNEGO), 128
Security Support Provider Interface
see SSPI
securityPackage property, provider tag, 143
SendCommand method
POP3 protocol, 429, 431
SMTP protocol, 426
senderEmail parameter
SMTPClientChannel, 446
SMTPServerChannel, 454
SendMessage method
encapsulating SMTP protocol, 426, 427
mapping protocols to .NET Remoting, 440
SendReplyMessage method, 440
SendRequestMessage method, 440
creating client-side sink and provider, 451, 452
SendResponseMessage method, 440
serializable objects
see also passing by value
ByValue objects, 25
passed between application domains, 492
problem with SoapSuds, 77
server-side asynchronous processing, 347
versioning, 242–245
servers require different versions, 256–273
SerializableAttribute class, 489
serialization
advantages of .NET Remoting, 12
BinaryClientFormatterSinkProvider, 502
BinaryServerFormatterSinkProvider, 501
brief description, 3
concerns regarding SoapSuds, 286
custom exceptions, 288
deserialization of object security, 93
ISerializable, 519
message objects through formatters, 329–330

servers requiring different versions, 266, 267
SoapClientFormatterSinkProvider, 503
SoapFieldAttribute, 515
SoapMessage, 521
SoapMethodAttribute, 516
SoapServerFormatterSinkProvider, 502
SoapTypeAttribute, 515
types of .NET Remoting, 25
Serialization namespaces
see under System.Runtime namespaces
SerializationException class, 521
BinaryFormatter version incompatibility, 309
SerializationInfo class, 520
custom exceptions, 289
SerializeSoapMessage method, 330
server applications
debugging hints, 303
deploying server application, 108
server assembly
multiserver configuration, 63, 64, 66
output independent of OneWay attribute, 58
output using asynchronous calls, 55
output using synchronous calls, 51
remoting application architecture, 14
sample remoting application, 15, 21
synchronous calls, 47
wrapped proxies, SoapSuds, 68
server objects
configuring in client configuration, 165
server-activated objects
see SAO
server-side channels, 453–462
server-side messaging, 333–338
server-side sinks, 354–355, 364–367
compression sinks, 360
configuration files, 370
connection using/not using compared, 370
EncryptionServerSink, 383
implementing server-side channels, 458–462
IServerChannelSink, 534
IServerChannelSinkProvider, 535
sink providers, 368
server-side sponsors, 203–209
server-side transport channel sinks, 426
ServerChannelSinkStack, 334, 460
ServerContextTerminatorSink, 337
ServerException class, 497–498
ServerObjectTerminatorSink, 337
ServerProcessing property, 392, 398
serverProviders tag
BinaryServerFormatterSinkProvider, 501
channel tag, 91, 202
formatter tag, 94, 95
HttpServerChannel, 506
server accepting authenticated requests, 146
sink providers, 349
ServerProxy property, 169
servers
BinaryServerFormatterSink, 503
BinaryServerFormatterSinkProvider, 501
client for second server component, 175
console window for final server, 177
creating server for remoting clients, 161–163
HttpServerChannel, 506
intermediary .NET Remoting server hosted in IIS, 173
lifecycle management, versioned SAOs, 237
output window for IIS hosted server, 177
registering POP3 server, 435
remoting events, 214
servers requiring different versions, 258, 260, 263, 264, 267, 268, 272
SoapServerFormatterSink, 504
SoapServerFormatterSinkProvider, 502
TcpServerChannel, 508
transfer runtime information with client, 209–213
versioning with interfaces, 247, 251, 252, 255
servers hashtable, 437, 438, 441, 444
ServerStartup class
published objects, 33
sample remoting application, 16
Singleton objects, 30
service installer
Windows service installer, 109
service tag, 85, 96, 172
Service-Oriented Architecture (SOA), 279, 281
ServiceBase class, 108
ServiceRequest method, 335
services
see also Windows services
ActivatedServiceTypeEntry, 494

ChannelServices, 500
EnterpriseServicesHelper, 516
LifetimeServices, 511
RemotingServices class, 493
TrackingServices, 517
WellKnownServiceTypeEntry, 495
Services namespace, 516, 518
session affinity
creating NLB clusters, 296
Session Ticket (ST), 127
SetAge method, 251, 252, 254
SetData method, 210
SetDefaultAuthenticationEntry method, 404, 407
SetSinkProperties method, 405
setValue method, 331
shared assemblies, 67
see also general assembly
.NET Framework versioning example, 229
creating server for remoting clients, 161
defining interfaces, 102
GAC, 227
interface definitions, .NET Remoting, 11
MSDN security, 140
multiserver configuration, 62
private assemblies compared, 227
servers requiring different versions, 260, 270
shared base classes, 67
shared implementation, 67
SoapSuds metadata, 68
versioning with interfaces, 246, 251, 253
shared base classes, 67
shared implementation, 67
shared interfaces, 11
shared storage, 299
signatures
delegates, 51
fingerprints, 225
symmetric encryption, 379
versioning with interfaces, 253
Simple Object Access Protocol
see SOAP
SingleCall objects
client output for, 29
registering, 28
scalable remoting rules, 280
server output for, 30
server-activated objects, 26, 28
wellknown tag, service tag, 97
SingleCall SAOs, 277
Singleton objects
allowing client access to, 83
changing base lifetime example, 188, 189, 191, 193
client output for, 31, 33
configuration files, 82
creating server for remoting clients, 163
lifetime management, 30
LifeTime property name extension, 195
registering, 30
RenewOnCallTime property name extension, 195
server output for, 32, 33
server-activated objects, 26, 30
SponsorshipTimeout property name extension, 195
threading, 32
versioning with interfaces, 248, 256
wellknown tag, service tag, 97
sink providers, 349–355, 367–369
changing sinks programming model, 406
channel tag, 91
client-side sinks, 367
reference for, 92
server-side sinks, 368
SinkProviderData objects, 407
sinks
BaseChannelObjectWithProperties, 539
BaseChannelWithProperties, 540
BinaryClientFormatterSink, 503
BinaryClientFormatterSinkProvider, 502
BinaryFormatter version incompatibility, 310, 409–413
BinaryServerFormatterSink, 336, 503
BinaryServerFormatterSinkProvider, 94, 501
business logic, 419
changing default remoting behavior, 359
changing programming model, 402–409
channelSinkProviders tag, 86
client-side sinks, 350–353, 361–364
implementing client-side channels, 449–453
ClientContextTerminatorSink, 332
compression sinks, 359–375
creating proxies, 323
CrossContextChannel, 336
DispatchChannelSink, 336
dynamic sinks, 332, 356–357

encryption, 380–385
HttpServerTransportSink, 335
IChannelSinkBase, 532
IClientChannelSink, 532
IClientChannelSinkProvider, 533
IClientFormatterSink, 534
IClientFormatterSinkProvider, 534
IMessageSink, 526
IServerChannelSink, 534
IServerChannelSinkProvider, 535
LeaseSink, 337
message processing, 531
message sinks, 321, 328–329
passing runtime information, 390–401
pluggable sink architecture, 402
SDLChannelSink, 335
server-side asynchronous processing, 347
server-side sinks, 354–355, 364–367
implementing server-side channels, 458–462
ServerContextTerminatorSink, 337
ServerObjectTerminatorSink, 337
sink chains, 349, 350, 455
sink providers, 367–369
SoapClientFormatterSink, 333, 503
SoapClientFormatterSinkProvider, 503
SoapServerFormatterSink, 336, 504
SoapServerFormatterSinkProvider, 94, 502
StackbuilderSink, 337
transport channels, 419
using custom proxy, 413–419
SMTP protocol, 422–423
asynchronous communication, 421
encapsulating, 426
SMTP Response code classes, 422
SOAP binding to, 424
transferring e-mail, 422
SmtpChannel class, 465–467
wrapping transport channel, 462–465
SMTPClientChannel class, 445–453
SMTPServerChannel compared, 454
wrapping transport channel, 462
SMTPClientTransportSink class, 450, 452
SMTPClientTransportSinkProvider class, 449
SmtpConnection class, 426, 438, 440
SMTPHelper class, 437, 438
smtpServer parameter
SMTPClientChannel, 446
SMTPServerChannel, 454
SMTPServerChannel class, 453–462
implementing server-side channels, 454, 455, 457
parameters, 454
SMTPClientChannel compared, 454
wrapping transport channel, 462
SMTPServerTransportSink class
creating server-side sinks, 459, 461
implementing server-side channels, 455, 456
sn.exe
.NET Framework versioning example, 228
key pairs, 226
SOAP (Simple Object Access Protocol)
binding to SMTP, 424
creating e-mail headers, 425
introduction, 7
moving messages through transport channels, 330
scalable remoting rules, 281
serializing messages through formatters, 330
when to use .NET Remoting, 279
SOAP formatter
HttpChannel, 504
HttpClientChannel, 505
links to web sites, 548
SOAP Trace Utility, 157
SoapAttribute class, 515
SoapClientFormatterSink class, 503
client-side messaging, 333
serializing messages through formatters, 330
SoapClientFormatterSinkProvider class, 503
SoapFault class, 521
SoapFieldAttribute class, 515
SoapFormatter class, 329
SoapMessage class, 521
SoapMethodAttribute class, 516
SoapOption enumeration, 516
SoapOptions property, 516
SoapParameterAttribute class, 516
SoapServerFormatterSink class, 504
server-side messaging, 336
SoapServerFormatterSinkProvider class, 502
configuring typeFilterLevel in code, 94
SoapSuds
calling, 68
client-side sinks, 400

concerns regarding, 286
creating CAOs, 34, 240
generating SoapSuds wrapped proxy, 69
generating source code with, 80
interfaces or, 100, 286
lifecycle management, versioned SAOs, 236, 239
non-wrapped proxy metadata, 72
problem with SoapSuds, 77–81
RemotingClientProxy, 517
SoapMethodAttribute, 516
SoapSuds generated nonwrapped proxy's source, 240
SoapSuds metadata, 82
shared assemblies, 68
wrapped proxies, 68
client assembly, 70
SoapTypeAttribute class, 515, 516
SPNEGO (security package negotiate), 128
sponsors
see also client-side sponsors
basic sponsor example, 197
typeFilterLevel changing security, 311
client-side sponsors, 196, 197
calling expired object's method, 198–203
ClientSponsor, 510
deserialization of object security, 93
InstanceSponsor_Lifetime, 205
InstanceSponsor_RenewOnCallTime, 205
ISponsor, 196, 509
LeaseManager, 196
lifetime management, 43, 196–209
LifetimeServices, 511
MarshalByRefObject, 185
registered sponsors, 93
remote objects, 196
remote sponsors, 203
server-side sponsors, 203–209
sponsorship
TTL lifetime management, 185
sponsorshipTimeout attribute, lifetime tag, 88
SponsorshipTimeout property, 186, 195
SSL encryption, 139
SSPI (Security Support Provider Interface), 128
differences versions 1.x and 2.0, 152
MSDN security samples, 140
new features also for v2.0, 140
ST (Session Ticket)
Kerberos authentication, 127
StackbuilderSink, 337
StartListening method, 455, 457, 464
STAT command, POP3, 424
state
client-activated objects, 34
creating server-side sinks, 459
creating Windows Forms client, 169
CurrentState property, 511
LeaseState enumeration, 511
links to web sites, 547
scalable remoting rules, 281
SingleCall objects, 28
StreamingContextStates enumeration, 520
static fields
scalable remoting rules, 281
StopKeepAlive method, 206
StopListening method, 464
storage
shared and local storage, 299
StreamingContext structure, 520
StreamingContextStates enumeration, 520
strictBinding attribute
BinaryServerFormatterSinkProvider, 501
formatter tag, serverProviders tag, 92, 95
strong names/naming, 225
.NET Framework versioning example, 228
CLR resolving assembly references, 227
creating strongly named assembly, 226
fingerprints, 225
GAC, 227
gacutil.exe retrieving, 235
versioning serializable objects, 242
versioning with interfaces, 247, 248, 250
strongly named assemblies
AssemblyKeyFile attribute, 227
AssemblyVersion attribute, 227
CLR locating, 227
creating, 226
lifecycle management, versioned SAOs, 234, 236
source file attributes, 226
suppressChannelData attribute, channel tag, 90
symmetric encryption, 376–380
synchronous calls, 46, 47
client assembly, 49
client-side sinks, 363
mapping, 421
proxies creating messages, 323
server-side sinks, 366

synchronous messaging, 441
SyncProcessMessage method
  handling asynchronous response, 346
  IMessageSink, 328
    asynchronous processing, 338, 339
    checking parameters in, 480
  passing runtime information, 390
  proxies creating messages, 323
  serializing messages through formatters, 329
  sinks using custom proxy, 415
system maintenance, 292
System.Runtime namespaces
  identifying for classes, 186
  Remoting, 491–499
  Remoting.Activation, 529–530
  Remoting.Channels, 499–504, 531–540
  Remoting.Channels.Http, 504–506
  Remoting.Channels.Tcp, 506–508
  Remoting.Lifetime, 508–511
  Remoting.Messaging, 512–514, 525–529
  Remoting.Metadata, 514–516
  Remoting.Proxies, 530–531
  Remoting.Services, 516–518
  Serialization, 518–521
  Serialization.Formatters, 521–523

T

tags
  see under configuration files
TCP channel attributes, 89
TCP channels
  see also channel tag
  creating console clients, 164
  creating server for remoting clients, 163
  cross-process on single machine, 276
  encryption, 375
  versioning with interfaces, 248
TCP connections
  HTTP requests, 292
  NLB clusters, 291, 292
  troubleshooting client behind firewall, 317
Tcp namespace, 506, 508
TcpChannel class, 506
TcpClientChannel class, 507, 539
TcpEx channel, 545
TcpServerChannel class, 508
TGT (Ticket Granting Ticket)
  Kerberos authentication, 126
ThreadPriority, 392, 397
threads
  asynchronous calls, 421
  changing default remoting behavior, 359
  ILogicalThreadAffinative interface, 209, 514
  mapping protocols to .NET Remoting, 440, 441
  passing runtime information, 391, 399, 401
  server-side sponsors, 206
  Singleton objects, 26, 32
threat modeling, 124
time
  see also lifetime management
  InitialLeaseTime property, 186
  InstanceSponsor_RenewOnCallTime, 205
  Just-In-Time debugging, 305
  leaseManagerPollTime attribute, 88
  leaseTime attribute, 88
  LeaseTimeAnalyzer method, 186
  RemotingTimeoutException class, 497
  RenewalTime property, 510
  renewOnCallTime attribute, 88
  RenewOnCallTime property, 186
  runtime information, 209–213, 390–401
  sponsorshipTimeout attribute, 88
  SponsorShipTimeout property, 186
Title property
  versioning serializable objects, 243, 245
trace
  SOAP Trace Utility, 157
TrackingServices class, 516, 517, 547
TransparentProxy objects
  client-side sinks, 353
  creating proxies, 322, 323
  proxies creating messages, 323
  proxies returning values, 324
  RealProxy, 531
  remoting with MarshalByRefObject, 59
transport channels, 421–468
  brief description, 321
  client-side transport channel, 445–453
  encapsulating SMTP/POP3 protocol, 426
  how messages work, 326
  moving messages through, 330–331
  preparing to use, 467
  server-side transport channel, 453–462
  sinks, 419
  SMTPClientChannel, 445–453
  SMTPServerChannel, 453–462

using SmtpChannel, 465
words of caution when developing, 468
wrapping transport channel, 462–465
transport headers
ITransportHeaders, 536
TransportHeaders method, 443
troubleshooting, 303–318
BinaryFormatter version incompatibility, 309–311
typeFilterLevel changing security, 311–313
client behind firewalls, 317–317
configuration files, 305–309
debugging, 303–305
multihomed machines, 315–317
using custom exceptions, 313–314
trusted subsystem, 178
try...catch blocks
one-way calls, client assembly, 56
TTL (time-to-live)
changing default lease time, 187
changing lease time on class by class basis, 187
expired TTL exception, 187
lifetime management, 185
sponsors, 196
TimeSpan properties, 186
tutorials
links to web sites, 543
type attribute
activated tag, client tag, 99
activated tag, service tag, 97
channel tag, 89
configuration file debugging, 307, 308
formatter tag, serverProviders tag, 92
provider tag, serverProviders tag, 92
wellknown tag, 83
client tag, 98, 99, 101
service tag, 82, 97
typed DataSets
concerns regarding SoapSuds, 287
TypeEntry class, 493, 494
typeFilterLevel attribute
BinaryClientFormatterSinkProvider, 502
BinaryServerFormatterSinkProvider, 501
changing security restrictions with, 311–313
formatter tag, 92, 93, 94
remoting events, 220
setting to full, 249
SoapClientFormatterSinkProvider, 503
versioning with interfaces, 248
TypeFilterLevel enumeration, 521
TypeName property, messages, 327
types
ActivatedClientTypeEntry, 494
ActivatedServiceTypeEntry, 494
content-type header, 409
GetRegisteredWellKnownClientTypes method, 101
InitTypeCache method, 100
IRemotedType interface, 103
IRemotingTypeInfo, 498
loadTypes attribute, debug tag, 87
object types, 13
RegisterActivatedServiceType method, 494
registering as remote, 99
SoapTypeAttribute, 515
TypeEntry, 493
versioning behavior, 95
WellKnownClientTypeEntry, 496
WellKnownServiceTypeEntry, 495
XmlTypeNamespace attribute, 241

U

/u parameter, 228
UDDI (Universal Description, Discovery, and Integration, 7
UDP broadcasts, 282, 283
UML diagram
multiserver configuration, 60
Unicode, 424
Unregister method
client-side sponsors, 200
server-side sponsors, 204, 207
unsafeAuthenticatedConnectionSharing attribute
channel tag, 90
Unwrap method, 492
UploadPerson method, 259, 261, 268, 271
Uri property, messages, 327
URIs
GetUrlsForUri method, 538
objectUri attribute, 97
url attribute, client tag, 83, 98, 99, 101
url tag, clientProviders tag, 402, 406, 409
UrlAuthenticationEntry class, 403, 406
UrlAuthenticationSink class, 404
UrlAuthenticationSinkProvider class, 407
UrlAuthenticator class, 402, 403
URLs
links to web sites, 541–548
parsing for e-mail address, 444

useAuthenticatedConnectionSharing attribute
channel tag, 90
useDefaultCredentials attribute, channel tag, 90, 138
useIpAddress attribute, channel tag, 90
User Datagram Protocol (UDP), 277

V

ValidationResult class, 20
value
AddValue method, 243, 261
ByValue objects, 13, 25
marshal by value object, 489
passing by value, 13
return values, 324
setValue method, 331
versioning
.NET Framework, 225–233
.NET Remoting, 233–245
advanced concepts, 246
application design, 273
AssemblyVersion attribute, 227, 237
client-activated objects, 240–242
component compatibility, 225
getSAOVersion method, 236
includeVersions attribute, 92
major/minor versions, 225
serializable objects, 242–245
servers require different versions, 256–273
server-activated objects, 233–239
strong naming, 225
versioning with interfaces, 246–256
versioning behavior, 95
Virtual Directory Creation Wizard, 117
virtual root, 117

W

WaitAndGetResponseMessage method, 441, 451
waitingFor hashtable, 437, 438, 440, 441
web application client
designer for creating back-end-based client, 170
web services
.NET Remoting or, 279
ASMX Web Services, 279
introduction, 7
links to web sites, 543, 547
servers requiring different versions, 257
web site references
links to web sites, 541–548
web.config file
.NET Remoting client configuration, 170
configuring ASP.NET client, 170
deployment using IIS, 116
lifecycle management, versioned SAOs, 235
remoting components hosted in IIS as clients, 172, 174
wellknown tag, 76, 82
client tag
attributes, 99
configuration file using attributes, 83, 100, 120, 139
service tag
anonymous deployment, 118
attributes, 96
configuration file using attributes, 98, 118
lifecycle management, versioned SAOs, 235, 238
WellKnownClientTypeEntry class, 496–497
WellKnownObjectMode enumeration, 497
WellKnownServiceTypeEntry class, 495–496
Whidbey, .NET Framework, 151
Windows authentication
enabling, 137
IIS authentication modes, 134
Windows event log, 110
Windows Forms applications
Windows Forms client, 167–169
security, 179, 181
Windows groups, IIS, 149
Windows Management Instrumentation (WMI)
creating NLB clusters, 298
Windows Network Load Balancing
see NLB
Windows NT challenge/response
see NTLM authentication
Windows services
debugging, 113
deploying server application, 108
hosting remote components, 110
installing service using installutil.exe, 112
integrating remoting in, 108
porting to, 108
security without IIS, 140

service installer, 109
starting from IDE, 111
WindowsIdentity, 130
WindowsPrincipal, 130, 131
wrapped proxies, 68, 69, 71
wrapper classes
accessing CallContext directly in code, 212
message contents, 327
WSE-DIME
links to web sites, 546

X

X-Compress header, 372
X-EncryptIV header, 380
X-REMOTING
creating e-mail headers, 425
mapping protocols to .NET Remoting, 443
XML-RPC, 7, 544
XmlNamespace attribute, 241
XmlTypeNamespace attribute, 241